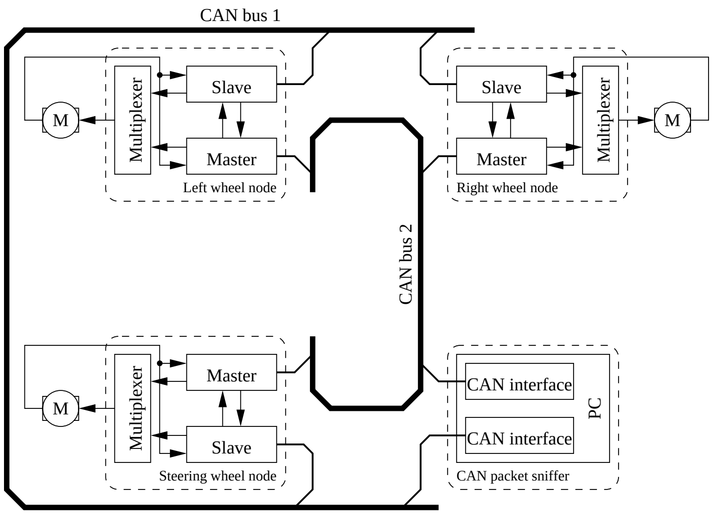
Aalborg University
Institute of Electronic Systems
Department of Control Engineering

Prototyping a Fault-tolerant CAN Bus Based Distributed Servosystem

Worksheets



Michael Skipper Andersen, Jørgen Friis, Niels Nørregård Hansen
Johnny Jensen, Rene Just Nielsen, Michael Pedersen

7. Semester • Group 731 • 2002

Contents

Chapter 1
Introduction

The objective of the project is to make a distributed servo system. This means the system, on the top level, must be able to make the two wheels and the steering wheel turn as if they were mechanically connected.

The system contains of the following parts:

- A steering wheel node.
- Two wheel nodes.
- A steering wheel.
- Two servo motors.
- A PC to monitor the bus traffic.

The nodes of the system are connected with double CAN bus as in figure 1.1.

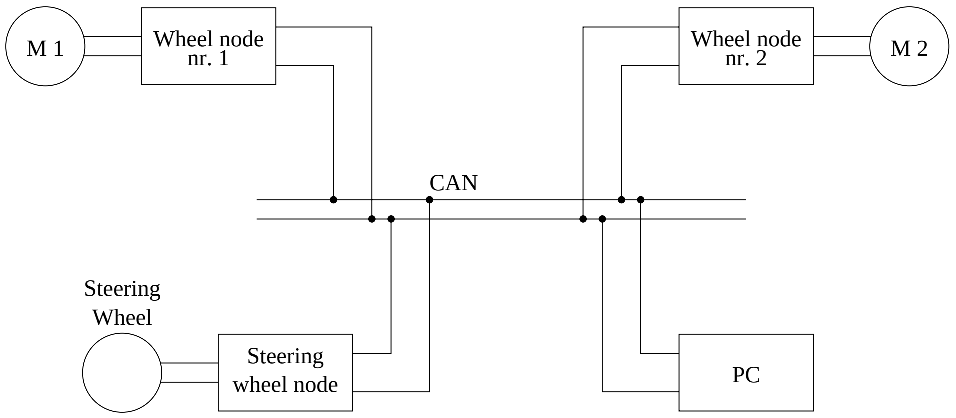


Figure 1.1: The setup of the steering wheel system.

The different parts of the system is described in the following way:

The three nodes: each consist of two PICs (microcontrollers) each are connected to a CAN bus. Further, the PICs are interconnected through the SPI (Serial Peripheral Interface) line in a master/slave configuration. The two PICs of a node perform basicly the same operations and they agree on sending the same messages on the CAN bus.

The steering wheel: Is of the force feedback type which means that it can “resist” if the two wheels can not be moved further.

The servo motors: Small DC motors that contain a potentiometer to feed back the position of the motor shaft.

The PC: Contains a card with two CAN drivers to monitor the bus traffic. It further contains software to print out the different CAN messages on the buses and to save the bus traffic in a text file.

Further the three nodes perform different top level tasks:

The steering wheel node: It collects the data from the two wheels and the steering wheel itself and decides the new positions of the wheels and steering wheel. Further it has to act on any alarm or error sent from the wheels.

The wheel nodes: Must feed back their actual position to the steering wheel node and report any error from the node.

Chapter 2

Analysis

2.1 CAN-bussen

Informationerne i dette afsnit beror på oplysninger fra den officielle CAN-specifikation fra Bosch [cdrom, can_spec.pdf] og [cdrom, MCP2510.pdf].

CAN-bussen (Controller Area Network) er en 1 Mbit/s seriel bus oprindeligt udviklet til brug i bilindustrien. Ideen er, at alle bussens knudepunkter kan sende og modtage på bussen og har et antal acceptfiltre, som gør, at de kun modtager data frames med bestemte identifiers.

CAN-protokollen benytter sig af CSMA/DCR[1].

Alle knudepunkterne lytter på bussen og er i stand til at registrere, om en frame er gyldig, dvs. fejlfri. Hvis blot ét knudepunkt registrerer en fejl i framen, vil denne begynde at transmittere error frames og afsenderen vil forsøge at sende framen igen. Hvis et knudepunkt gentagne gange transmitterer eller modtager en fejlbehæftet frame, vil den efter et bestemt antal forsøg ekskludere sig selv fra netværket.

CAN-bussen kan benytte 3 protokolversioner:

- *type 2.0A* kan sende og modtage 11-bit-identifiere.
- *type 2.0B passiv* kan sende 11-bit-identifiere og modtage 29-bit-identifiere.
- *type 2.0B aktiv* kan sende og modtage 29-bit-identifiere.

2.1.1 Frame-typer

Der er 4 typer frames:

- *Data frames* bruges til at sende op til 8 bytes data.

[1]Carrier Sense Multiple Access with Deterministic Collision Resolution.

- *Remote frames* bruges til at forespørge data til afsenderen af remote framen.
- *Error frames* transmitteres af et knudepunkt, der har opdaget en fejl i transmissionen.
- *Overload frames* udsætter transmissionen af den følgende data eller remote frame.

Data frame

På figur 2.1 ses en data frame, som er opdelt i følgende områder:

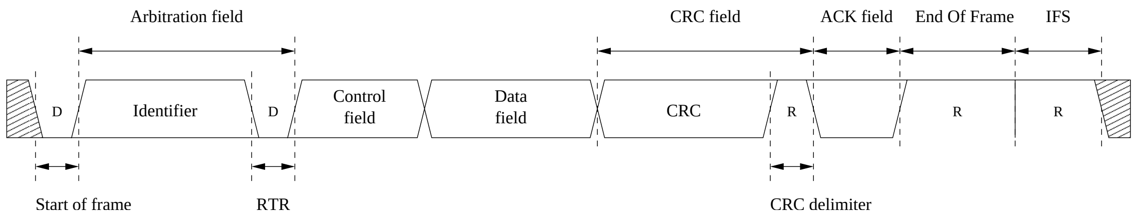


Figure 2.1: Data frame.

Start Of Frame er en "dominant"[2] bit, som indikerer starten på en frame.

Arbitration field består af en identifier og en RTR-bit (Remote Transmission Request). Der skelnes mellem 11- og 29-bit identifiere afhængig af, hvilken protokolversion der benyttes. Hvis flere knudepunkter transmitterer en frame på bussen på samme tid, vil identifieren med den laveste talværdi, i henhold til princippet om recessive og dominante bits, få lov at fortsætte transmissionen. De andre knudepunkter lytter på bussen og genudsender deres respektive frames, når den igangværende transmission er færdig (se figur 2.2).

RTR-bit'en indikerer forskellen på en data frame og en remote frame. I en data frame er den dominant, mens den er recessiv i en remote frame.

Control field er et 6-bit område, der angiver, hvor mange data-bytes, der transmitteres. Kun talværdierne fra 0–8 benyttes.

Data field indeholder fra 0–8 bytes data fra afsenderen. Det benyttes kun i en data frame.

CRC field (Cyclic Redundancy Check) består af en 15-bit kode til beregning af fejl i transmissionen og en recessiv CRC delimiter-bit. Hvert knudepunkt læser og sammenholder CRC-koden med deres egne beregninger.

[2]Der skelnes mellem "dominante" og "recessive" bits. Hvis en dominant og en recessiv bit lægges på bussen på samme tid, vil den resulterende værdi blive dominant.

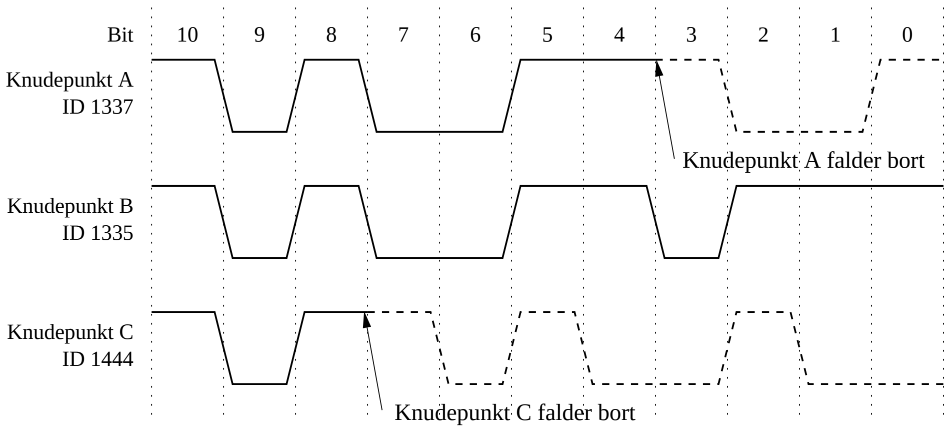


Figure 2.2: Bit-arbitrering. Knudepunkt B får lov at udsende data på CAN-bussen.

ACK field (Acknowledge) består af et ACK slot og en ACK delimiter-bit. Knudepunktet, der har afsendt framen sender et recessivt ACK slot-bit, mens alle øvrige knudepunkter, der har modtaget en fejlfri frame (beregnet udfra CRC-koden) transmitterer en dominant bit. Efter ACK slot'en udsender afsenderen en recessiv ACK delimiter-bit.

End of frame består af syv recessive bits, der indikerer slutningen af en frame.

IFS efter tre recessive IFS-bits (Interframe space) er bussen tilgængelig for nye afsendere.

Remote frame

På figur 2.3 ses en remote frame.

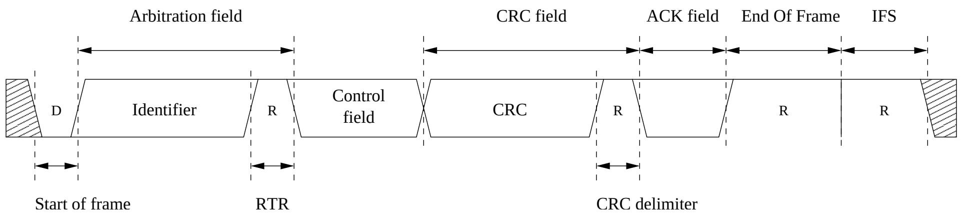


Figure 2.3: Remote frame.

Et knudepunkt, som ønsker at modtage data, kan initiere afsendelse af disse ved at sende en forespørgsel i form af en remote frame. Den eneste forskel på en remote frame og en data frame er, at remote framens RTR-bit er recessiv og at den ikke indeholder nogen data-bytes.

Identifieren i den forespurgte data frame er den samme som remote framens.

Error frame

Frame-segmenterne Start of frame, arbitration, control, data og CRC field undergår en kodning kaldet bit stuffing. Hvis afsenderen opdager 5 ens på hinanden følgende bits, indsætter den selv én komplementær bit og modtagerne sørger selv for at fjerne den igen.

En error frame transmitteres, når et knudepunkt opdager en fejl i den igangværende transmission. Da denne starter med 6 ens på hinanden følgende bits, som enten bryder reglen for bit stuffing eller det faste mønster i ACK- eller end of frame-sekvensen, kan den afsendes på et vilkårligt tidspunkt i framen.

Error framen vist på figur 2.4 består af 6 error flag, 0–6 error echo flag og 8 error delimiter flag.

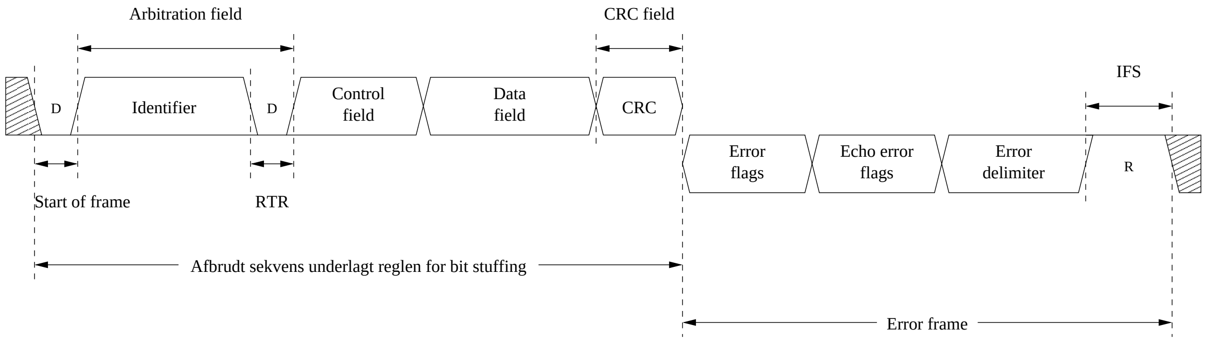


Figure 2.4: Error frame, der afbryder en igangværende data frame.

Der skelnes mellem *error active*- og *error pasive*-knudepunkter. Et error active-knudepunkt vil afbryde den igangværende frame med 6 dominante error flag. Hvis ikke samtlige error active-knudepunkter opdager fejlen på bussen, vil de andre registrere en bit stuffing fejl i løbet af højst 6 bits (de 6 error flag) og begynde at transmittere 6 error flag resulterende i 0–6 echo error flag (se figur 2.5).

Et error passive-knudepunkt vil derimod ikke afbryde den igangværende transmission, men tilkendegive at det har opdaget en fejl ved at udsende 6 recessive error flag.

En error frame afsluttes med 8 recessive error delimiter-bits.

Overload frame

En overload frame kan transmitteres som følge af to betingelser:

- Hvis overload framens afsender har brug for en tidsforsinkelse, inden næste data eller remote frame afsendes.

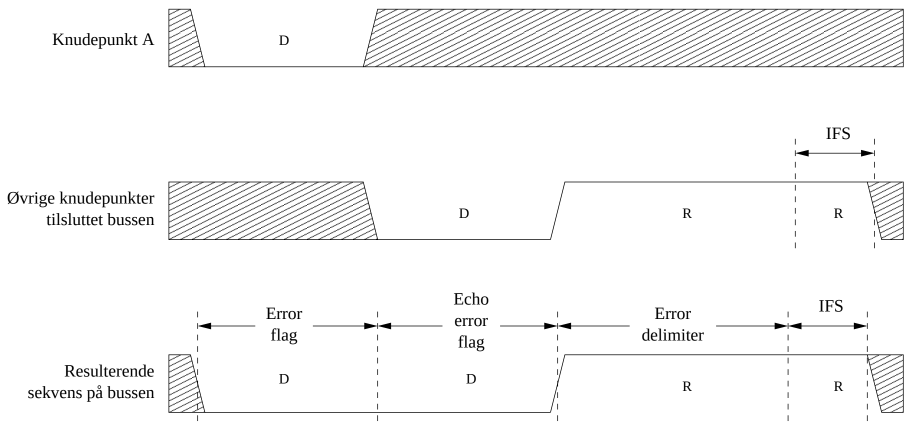

Knudepunkt A
D
IFS
Øvrige knudepunkter tilsluttet bussen
D
R
R
Error flag
Echo error flag
Error delimiter
IFS
Resulterende sekvens på bussen
D
D
R
R

Figure 2.5: Når knudepunkt A har transmitteret en error frame opdager bussens øvrige knudepunkter dette og udsender echo error frames.

- Hvis der opdages et dominant bit i IFS-feltet.

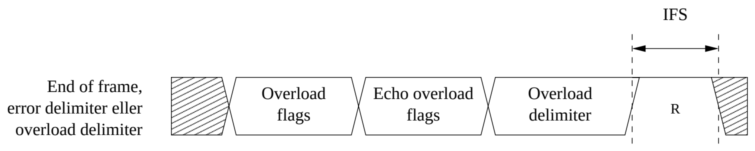


Figure 2.6: Overload frame.

Den har præcis samme format som en error frame (se figur 2.6), men transmitteres først efter end of frame, error delimiter eller overload delimiter. Der kan højst transmitteres 2 overload frames efter hinanden.

2.1.2 Fejlhåndtering

Hvert knudepunkt indeholder to 8-bit-registre til fejlregistrering - Transmit Error Count (TEC) og Receive Error Count (REC) - og indholdet af disse bestemmer, hvilken af følgende tre tilstande knudepunktet befinder sig i:

Error active hvis værdien af både TEC og REC er mindre end 128.

Error passive hvis værdien af TEC eller REC er mellem 128 og 255.

Bus off hvis værdien af TEC eller REC overskrider 255.

For hver vellykket modtagelse dekrementeres REC og TEC med 1. En transmissionsfejl inkrementerer TEC med 8 og en modtagefejl inkrementerer REC med 1. Således vil et knudepunkt, der gentagne gange transmitterer fejlbehæftede frames hurtigt blive sat i bus off-tilstand.

Et knudepunkt i bus off-tilstand vil blive bragt tilbage i active error-tilstanden og få nulstillet TEC og REC, når det har registreret 11 på hinanden følgende recessive bits 128 gange på bussen.

2.2 Introducing Fault Tolerance

This section specifies the different actions made to introduce fault tolerance in the different nodes of the steering wheel system. The strategies proposed here are based on that the interconnection of the different nodes is made with a CAN BUS network and therefore follows the CAN BUS standard (ISO 11898???). That means that the handling of errors on the actual transmission line (that is CRC errors etc.) is not described here, but is assumed handled by the fault tolerance features in the CAN BUS standard.

2.2.1 Hardware Fault Tolerance

The basic in the hardware fault tolerance strategy in the steering wheel system nodes is replication. This means that in each node, two PICs run in parallel on two separate CAN-busses. This means that the system will continue to operate if one of the CAN lines breaks. Before the PICs start to use the received data for computations, the data from the two lines are compared, and the PICs agree on which data to use. On the other hand no replication is used on supply voltages or alike, which means that faults of this type is not considered.

2.2.2 Software Fault Tolerance

On the software level the two PICs of a node use handshakes to ensure that they are both running and they perform the right computations. This means that they compare the results of their computations before the results are used further. Further a reasonability check is performed on calculated values and received messages. If one PIC calculate an invalid result, this PIC will receive the valid result from the other PIC. If all data is found invalid, the node will use the latest valid data. This is done from the philosophy that it is better to use old valid data, than new invalid data! If both PIC have invalid data they must send a CAN error message with a double error flag.

The method to introduce fault tolerance into the three nodes of the steering system is done through the following steps (source: Copy from Roozbeh):

- Error detection.
- Damage confinement.
- Error recovery.
- Fault treatment and continued service.

The probability of errors in logic components such as the multiplexer and CAN drivers is very low, in according to communication errors etc. so it's assumed that these errors don't occur in this system. Of the same reasons it's also assumed that the connections between the components don't fail.

2.2.3 Error Detection

The errors in a node falls into three different groups:

1. Errors on the communication line.
2. Runtime errors in the PIC program.
3. Errors in the peripheral devices of a node.

The errors covered in 1 are all handled autonomously by the CAN modules in the PIC.

Runtime Errors in the PIC Program

Errors in the data extracting code: The two PICs "shake hands" after receiving a data frame, and compare the received data.

Detection of deadlocks in the program: The two PICs "shake hands" after central parts of the code has been executed.

Detection of errors in the calculations: The two PICs "shake hands" after the calculation of control signals.

Errors in the Peripheral Devices of the Node

Error in the sampling or potentiometer: The two PICs agree on the sampled value and make a range check on the result.

2.2.4 Damage Confinement

The concept of the damage confinement used in this context is a firewall concept, which means the following:

- At errors where the two PICs compare their data (or received messages) illegal data is discarded when it is detected and overwritten with valid data.
- Illegal data is discarded under range check.

2.2.5 *Error Recovery*

- After comparation of data, the data that is decided to be faulty is discarded and overwritten by valid data.
- The same is done at range check with one addition. If both sets of received data (or calculated value) are considered to be in the invalid range a retransmission/recalculation is requested together with sending a CAN-error frame.
- When a deadlock is detected the PIC that is alive will try to reset the dead PIC (and send a CAN-error message to the steering wheel that this has happened). This is continuing, even if the the dead PIC has not responded in a long time.
- When the potentiometer is detected to be faulty, the PICs try to sample the potentiometer again. If the new value still is faulty the current wheel position is maintained and an alarm is transmitted on the CAN-BUS.

2.2.6 *Fault Treatment and Continued Service*

- Errors in the potentiometer can't be recovered so when this error occur nothing can be done, and the system is out of function.
- After a reset of a PIC normal operation is tried, it means that the communication between the PICs are tried again.
- If a transmission line is faulty both PICs will still be operational. This means that the comparation of CAN messages will not be performed. On the other hand both PICs will calculate control signals and sample the potentiometer, so in this part the PICs still check each other.

2.3 *Motor Control Rights*

In connection with using two PICs to control one motor, an important question has to be answered: *Which of the two PICs has the right to apply the control signal on the motor?* This can be rephrased to which PIC may control the MUX and thereby the motor? It also has to be decided which PIC has the right when both PICs claim the right to the motor. This is further complicated in the situation where faults are introduced. To answer this question, it is first assumed that only one PIC can be faulty, that is a single failure approach is assumed.

In the following three different solutions to the problem is purposed and then discussed. This will serve as an argumentation for the use of one of them in the steering wheel system. The proposals are:

1. Use an external micro controller to to decide which system that has to operate the motor

2. Use external logical gates

3. Use the internal interrupts of the PICs

Use an External Microcontroller

An external microcontroller can be inserted in the system to decide which PIC that will have the right to control the motor. The external microcontroller must receive signals from both PICs and decide which PIC that has to control the motor by changing the multiplexer. A diagram is shown in figure 2.7

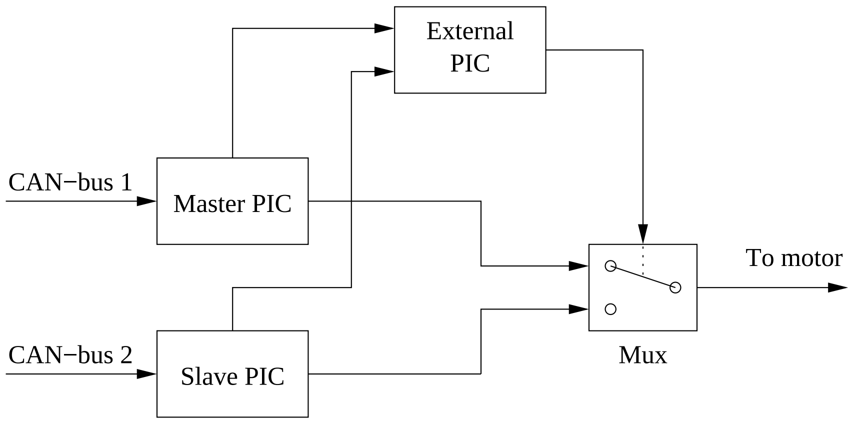


Figure 2.7: An external PIC controls the multiplexer.

Use of External Logical Gates

In this solution the two PICs are connected to the MUX control pin through some logical gates instead of a microcontroller. This can be more reliable because the probability of errors in logical gates is lower than in a microcontroller.

Use the Internal Interrupts of the PICs

Using the internal interrupts means that one PIC always controls the MUX. Only one PIC (the master) is connected to the MUX, and the master may as default apply control signals to the motor. The only way the slave can gain the right to apply a control signal is by generating an interrupt on the master, that hands over the motor control to the slave. The interrupt is generated through a dedicated connection between the master and slave, where the interrupt is triggered on a rising edge. The configuration is shown in figure 2.8.

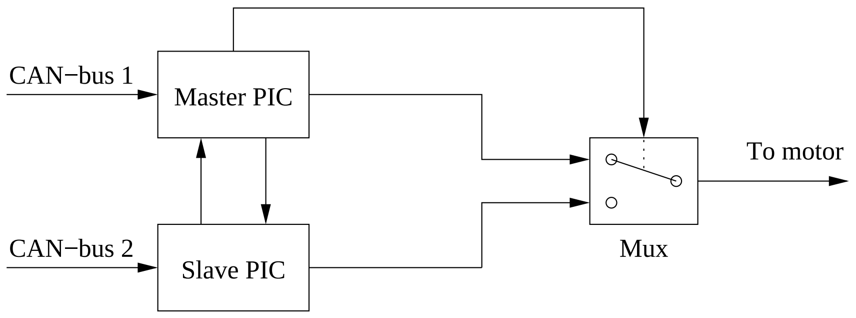


Figure 2.8: One PIC controls the multiplexer and the other can get the right through an interupt.

Discussion

The first proposal might seem as the obvious solution. The deciding PIC can be simple only needing few I/O ports and no other facilities. It will need a simple program to decide which of the two other PICs that can control the motor. On the other hand it introduces yet another unit that has the possibility of ending up in a deadlock. Further since the operation of the third PIC is simple, it might be better to use some simple logic due to the lower probability of errors in the simple logic gates.

The second proposal moves the decision about which PIC have the right to control the motor away from the two PICs and into a simple logical circuit. This indicates that even if the PICs contains faults, the right decision will be made after all. But by using gates, it is required that the output pins of the PICs are in a defined state that the gates recognize. If a PIC has crashed it is not known if these output pins will be in a high or a low state. This yields both for the master and the slave PIC and it makes it very difficult to design a good logical circuit to decide if it is the right PIC that controls the motor. Who has the right to overrule who???

The third solution is based on the assumption that it is always possible to generate an interrupt. This is taken as a fact since the detection and handling of interrupts are handled on the hardware level of the PIC (see data sheet!). Like in the second proposal, the master is assigned to the motor as default, but here it is also the master that controls the MUX. If the master should end in a deadlock, the slave will take the MUX control through the interrupt. If the slave contains a fault that holds the interrupt control pin high the slave will get the control over the motor. This will only happen in a short time, since the master as default will have the motor control when it is running and the interrupt is only generated on rising edge of the signal. Therefore the master will retain the motor control and the system continuous to operate.

Based on the previous arguments it is chosen to use the third solution in the steering wheel system. The main argument is that it is a simple solution that does not need additional components, and further it is considered more reliable. It further holds the possibility that the interrupt could contain a reset instruction to try to bring the master to life again after a deadlock.

Chapter 3

Demands

3.1 Protocol Description

The communication from the steering wheel node (SWN) to the two wheel nodes (WN) goes off with a fixed frequency. Whenever a wheel node has intercepted a reference value it attempts to control the wheel to this positions and hereafter responds with its current position to the steering wheel node.

The SWN sends reference values (refValMot1 and refValMot2) to the WNs that respond with their current values (curValMot1 and curValMot2). Besides, the SWN holds two variables (curValSw and refValSw) containing the current steering wheel position and the steering wheel reference value.

The two PICs in each node must always transmit the same contents on the CAN bus at approximately the same time.

If a node realizes that one of the CAN lines has been dead a certain number of consecutive times it should broadcast an alarm frame (a high-priority CAN frame) that the other nodes can react in correspondence with.

The protocol can be described with the following pseudo-code cases:

Case A

Steering wheel and wheels are all in the same position.

```
if (curValSw==curValMot1==curValMot2) {
  Do nothing. Everything is fine.
}
```

Case B

One of the wheels cannot go in the same position as the steering wheel. First, the system must then ensure that both wheels are in the same position that is, the wheel that is in the same position as the steering wheel should obtain the same position as the other (immovable) wheel. Hereafter the steering wheel must obtain the position of the wheels.

```
if (curValSw==curValMot1 && (curValMot1!=curValMot2)) {
  refValMot1=curValMot2
  refValSw=refValMot1
}
```

likewise for the other wheel:

```
if (curValSw==curValMot2 && (curValMot1!=curValMot2)) {
  refValMot2=curValMot1
  refValSw=refValMot2
}
```

Case C

The driver turns the steering wheel. The wheels must change their positions.

```
if (curValSw!=(curValMot1==curValMot2)) {
  refValMot1=curValSw
  refValMot2=curValSw
}
```

Case D

The steering wheel and the wheels are all in different positions. If this occurs the steering wheel and wheel 1 will attempt to adjust their positions to wheel 2. If they do not succeed within 5 consecutive samples the steering wheel and wheel 2 will attempt to adjust their positions to wheel 1.

3.1.1 CAN Messages

The CAN identifiers are chosen such that the the SWN has a higher priority than the WNs and the alarm frames have highest priority.

Address	
Steering wheel	00000001000
Wheel 1	00000010000
Wheel 2	00000100000
Alarm	00000000000

```
if (curValSw!=curValMot1!=curValMot2) (less than 5 consecutive sample times) {
  refValMot1=refValMot2
  refValSw=refValMot1
} else {
  refValMot2=refValMot1
  refValSw=refValMot2
}
```

When a WN reference value is changed or updated the following CAN frame is sent from the SWN.

Address	2 Bytes
Mot*	refValMot*

The same frame format is used when the WNs send their current values to the SWN.

Address	2 Bytes
Sw	curValMot*

The 2 data bytes contain the 10-bit value from the A/D converter, the first byte holding the 2 most significant bits and the other one holding the remaining 8 bits.

When an insolveable fault has occured an alarm frame is broadcasted. It has the following format.

Adresse	0 Bytes
Alarm	alarmType

The alarmType can specify a request for a retransmission of data or indicate that several insolveable errors have occured, e.g. when 10 consecutive wrong A/D converter value have been detected.

3.1.2 Alarm Handling

When an alarm frame is transmitted, the following data byte is send:

It is send as an integer in the program. The different flags indicate the following:

Doomed: Indicate if the alarm sender is not able to receive CAN or ADC values the last five times and therefor can not operate at all. The system ends it operation.

ADCerror: Indicate if the alarm sender has trouble sampling.

CANerror: Indicates if the alarm sender has received corrupted data.

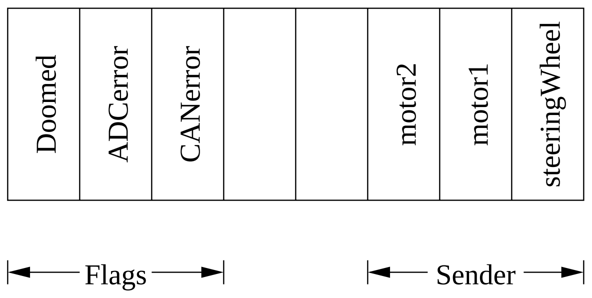


Figure 3.1: The alarm byte.

motor2: The alarm sender is motor 2.

motor1: The alarm sender is motor 1.

steeringWheel: The alarm sender is the steering wheel.

3.2 General Cases

Here follows som general cases the system has to respond to like described.

Case 1: Either master or slave receives a CAN-messages with corrupted data

If master or slave on a node receives a CAN message where the data is outside the valid range the PICs on the node must use the received CAN-message on the other bus.

Case 2: Both master and slave receives a CAN-messages with corrupted data

If both master and slave on a node receives a CAN message with data outside the valid range they must use the old reference values or motor position values and then send a CAN message to the deliver of the invalid message with a doubleerror flag set. This must lead to a retransmission of the former corrupted data.

Case 3: Either master or slave receives a CAN message and the other don't

If master or slave on a node doesn't receive a CAN message and the other do, the data from the recieved CAN message is used if it is inside the valid range of data. Otherwise the old data is used and a CAN error message is send with information about the doubleerror. If a doublerror is sent ten times consecutive, a CAN error message with information about we are doomed is send.

Case 4: Steering wheel is disconnected from the CAN bus

If a wheel node never receives new data on a CAN bus, it must continue to use the old data.

Case 5: Either master or slave is stuck somewhere in software

If master or slave on a node is stuck somewhere in the software, the other PIC on the node must reset the stucked PIC to make it operate normal again, and send a CAN message that tells that it has reset the PIC.

Case 6: Either master or slave samples invalid motor position data

If master or slave on a node samples invalid motor position values they must use the data from the one who samples the valid data to calculate the control signal.

Case 7: Either master or slave receives a CAN message with wrong data and the other receives nothing

If master or slave receives corrupted CAN data and the other does not receive anything on the CAN bus, the wheels do not know where to go. They must therefore stay in the position where they last have received valid data.

Chapter 4

Hardware

This section gives an overview of the hardware.

4.1 The Three Nodes Physical Description

Each node consists of two interconnected PICs. They are both connected to the same motor drive circuit but only one at a time is allowed to send a control signal to the motor. Therefore the two PICs are connected to a multiplexer (MUX) controled by the master PIC. Figure 4.1 shows the interconnections of the master and slave PIC and the pins are explained in table 4.1.

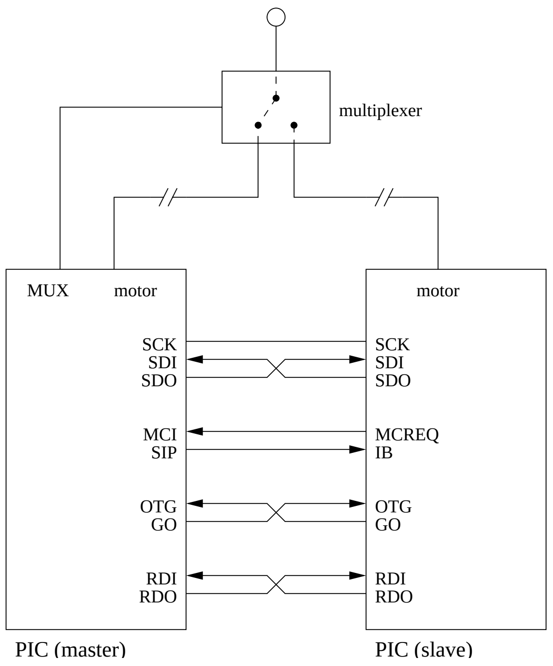


Figure 4.1: The interconnection of the master and slave PIC.

Pin	Description
RDO	(Ready Out) is asserted by a PIC to indicate that it is ready.
RDI	(Ready In) indicates that the other PIC is ready.
SCK	(Serial Clock) SPI Clock.
SDI	(Serial Data Input) SPI data input.
SDO	(Serial Data Output) SPI data output.
motor	The direction and PWM outputs from the PICs.
MUX	the multiplexer select output from the master PIC.
MCREQ	(Motor Control Request) Request signal to have the master negating MUX and letting the slave control the motor.
MCI	(Motor Control Interrupt) Interrupt from the slave wishing to control the motor.
SIP	Slave Interrupt Pin
IB	Interrupt Pin (actually IP, but IB sounds cooler..)

CHAPTER 5

PID CONTROLLERS FOR THE MOTORS

5.1 *Motor Modulation*

In general a DC-motor can be described by a first order transfer function from input voltage to output angular velocity. Because the angular position is measured on the motor using a potentiometer an integrator is included in the motor model. The potentiometer gives an output voltage from 0–5 V. This motor model is illustrated in figure 5.1.

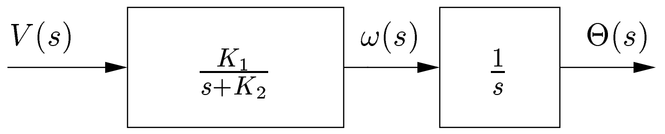


Figure 5.1: The motor model that describes the relationship between the input voltage and the output voltage delivered by the potentiometer.

In order to make a model of the motors used in the system, the motor placed in the steering wheel and the two position motors placed at each wheel, a couple of measurements on the motors were made.

The test setup for the motors is the actual hardware build for the motors.

On the oscilloscope both the input voltage to the motor and the output voltage from the potentiometer is saved in an .csv-file. This procedure was done for a wheel motor and the steering wheel motor.

The saved data for the motors was afterwards used to estimate the parameters for the motors. This was done by using the MATLAB$^{\text{TM}}$ program SENSTOOLS . Figure 5.2 shows the fit obtained on a wheel motor.

$$G_{\text{Wheel}}(s) = \frac{\Theta(s)}{U(s)} = \frac{0.0874}{s(s+14.3053)} \tag{5.1}$$

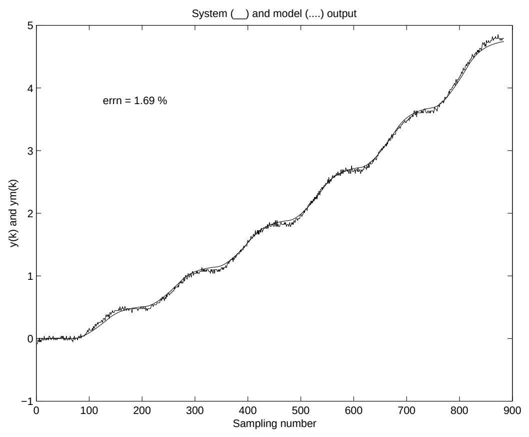


Figure 5.2: The parameter fit obtained by SENSTOOLS for the two positioning motors.

The same procedure is used on the steering wheel motor, and the transfer function is found to be:

$$G_{\text{Steering Wheel}}(s) = \frac{\Theta(s)}{U(s)} = \frac{0.1044}{s(s+36.0063)} \tag{5.2}$$

5.2 Controller Design

In order to control the positions of the motors a PID controller is used for each of them.

In continuous time the configuration shown in figure 5.3 is used.

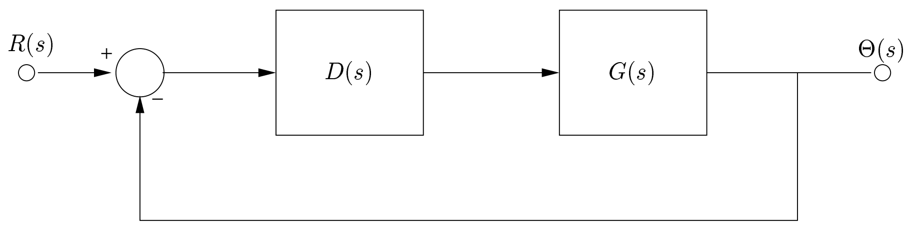


Figure 5.3: The control loop.

Where $D(s)$ is the transfer function for the controller and $G(s)$ is the transfer function for the respective motor.

For a PID-controller $D(s)$ is given by:

$$D(s) = K_p \cdot (1 + \frac{1}{T_i s} + T_d s) = K_p \left(\frac{T_d T_i s^2 + T_i s + 1}{T_i s} \right) \tag{5.3}$$

The demands for the controller are:

- No overshoot.
- Rise time less than 78 ms.

In order to fulfill these demands the MATLAB$^{\text{TM}}$ functions `rlocus` and `rlocfind` is respectively used to draw root loci and to determine the proportional gain.

The Wheel

In section 5.1 the following transfer function for the wheel motor was obtained:

$$G_{\text{Wheel}}(s) = \frac{\Theta(s)}{U(s)} = \frac{0.0874}{s(s+14.3053)} \tag{5.4}$$

From the transfer function for the wheel it is seen that it has pole in 0 and one in -14.3053.

From the PID a pole in 0 is introduced, and two zeros that can be places freely. Because the demand specifies that the system have to be stable and without overshoot the two poles have to be moved from the origo and into the left half plane on the real axis.

In order to make the poles move in that way it is necessary to place the two zeros from the PID as two distinct real zeros in the left half plane. In order to make the zeros real the numerator of $D(s)$ in equation 5.3 must have a positive discriminant which means:

$$T_i^{-4} \cdot T_i \cdot T_d \cdot 1 > 0 \quad \Leftrightarrow \quad T_i > 4T_d \tag{5.5}$$

Now the placement of the zeros is considered and there are three possible ways the this can be done:

- Both zeros are placed between the double pole in zero and the pole in -14.3053. The root locus for this situation is shown in figure 5.4(a)
- One zero is placed between the double pole in zero and the other is placed outside the pole in -14.3053. The root locus for this situation is shown in figure 5.4(b)
- Both zeros are places outside the pole in -14.3053. The root locus for this situation is shown in figure 5.4(c)

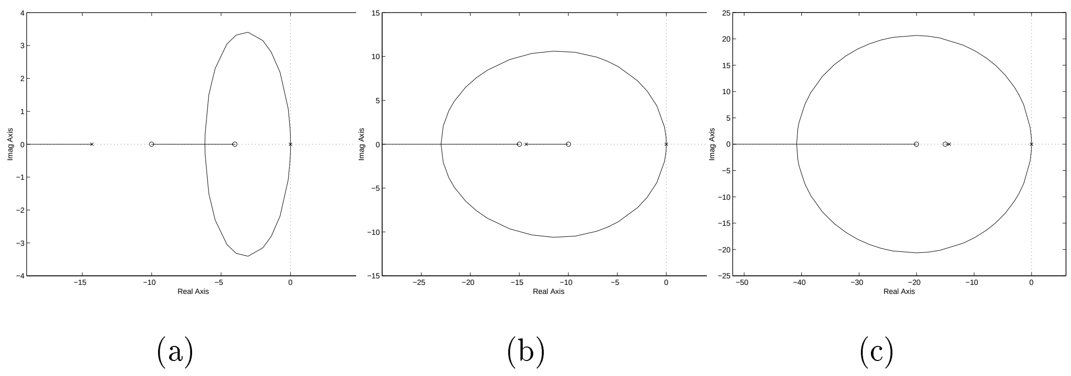


Figure 5.4: (a)Shows the root loci of case 1. (b) Shows the root loci of case 2. (c) shows the root loci of case 3.

As it is seen in the figure the distance the poles have to travel before they reach the real axis increase as the zeros are moved longer into the left half plane. And as the distance increase the required gain K_p to move the poles that distance is increased too.

Therefore it is chosen to place the zeros as in case 1. Because the control signal for the motor (the output delivered from the controller) is limited to ± 1024 the gain have to be held small. Therefore it is chosen to place one zero near the imaginary axis and the other some way from there. It Is chosen to use $T_i = 100$ and $T_d = 0.1$ which places the zeros in $z_1 = -0.01$ and $z_2 = -10.09$. This gives the root loci shown in figure 5.5.

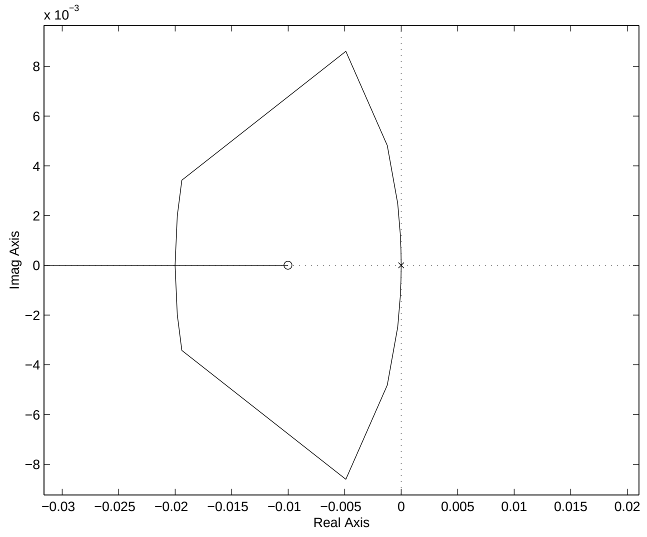


Figure 5.5: Root locus for the wheel.

By the use of the MATLAB™ function `rlocfind` K_p is found to be 1072.5 in order to make the poles real and fast. But if the rise time is measured from a step response of the system it is seen that the rise time demand is not fulfilled.

This is shown in figure 5.6.

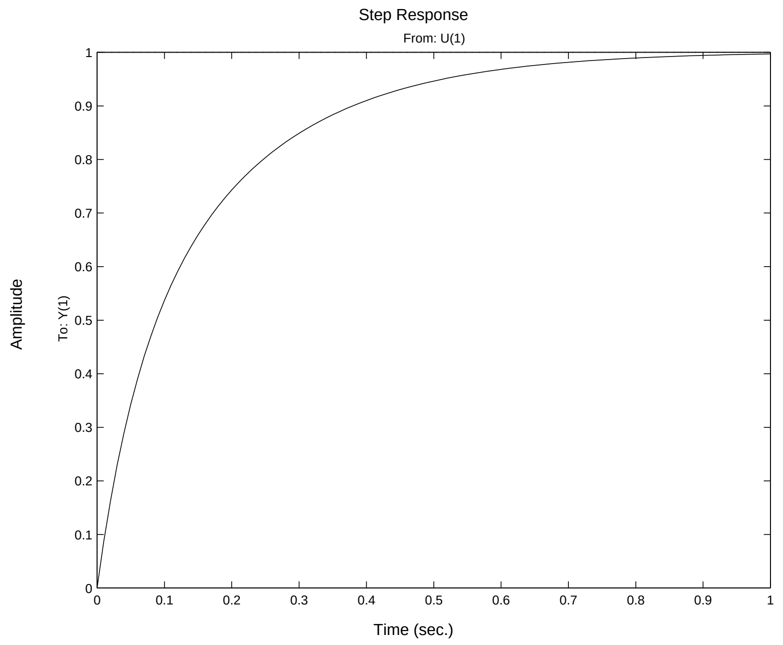


Figure 5.6: The step response of wheel motor and PID controller.

With $K_p = 1072.5$ the poles in the system are placed in, $p_1 = -18.66$, $p_2 = -5.01$ and $p_3 = -0.01$. Which means that by increasing K_p wouldn't make the system introduce any overshoot cause the poles would stay or the real axis.

Because the gain K_p already is considerably large in respect to the control signal limit of 1024 then it is known that the system won't function properly if the gain is made much larger when this saturation is inserted.

Therefore it is chosen not to tune the controller further before the discrete model of the system with the saturations is made. This is done later in this chapter.

The Steering Wheel

Same procedure as used for the wheel control system is used to design a controller for the steering wheel.

In section 5.1 the following transfer function for the steering wheel motor was obtained:

$$G_{\text{Steering Wheel}}(s) = \frac{\Theta(s)}{U(s)} = \frac{0.1044}{s(s + 36.0063)} \tag{5.6}$$

For the steering it is also chosen to use $T_i = 150$ and $T_d = 0.1$. Hereby the root locus shown in figure 5.7 is obtained.

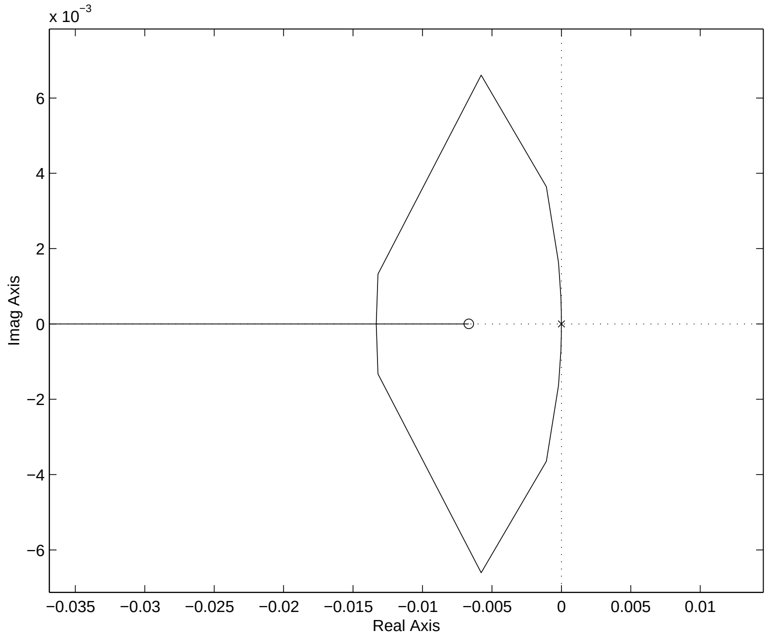


Figure 5.7: The root locus for the steering wheel motor with $T_i = 150$ and $T_d = 0.1$.

By using the MATLAB™ function `rlocfind` K_p is found to be 3000. Hereby the step response shown in figure 5.8.

Again no further adjustments of the controller is made before a discrete simulation model is made.

5.3 Discretization and simulations

Because the gains (K_p) for both type motors are quite large and the control signal is limited to be a maximum of 1024 it must be assumed that the control signal saturates and results in a slower system and with a considerable overshoot if no compensation for the saturation is inserted.

In order to determine the size of the overshoot and the rise time, when this saturation is inserted in the system, SIMULINK™ is used.

Because the controller has to be implemented in a computer a discrete controller is needed.

Because the system, that has to implement the controllers, uses analog to digital (A/D) converters the system is described by a zero-order hold (ZOH) discretization. And therefore the controllers are found by using ZOH.

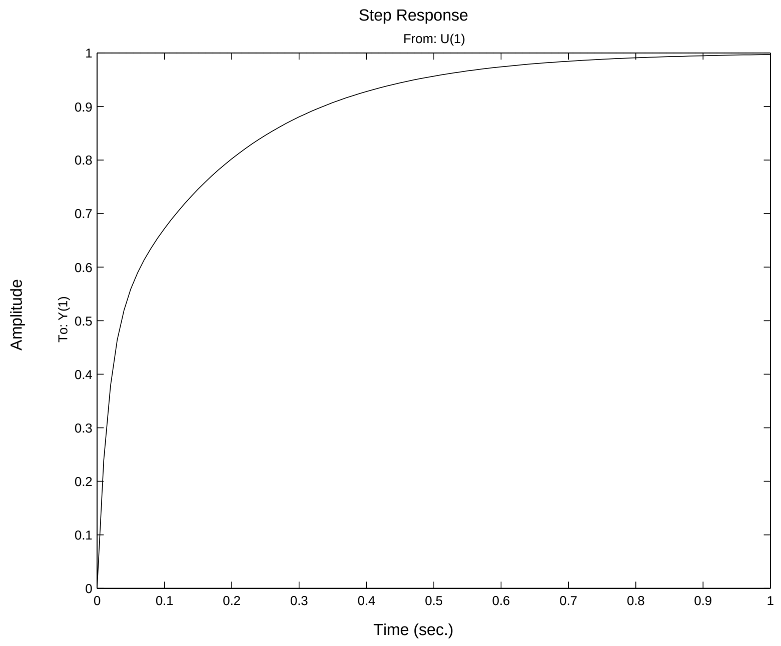


Figure 5.8: The step response of the steering wheel motor and controller system.

5.3.1 ZOH Discretization

The continuous-time controller transfer function $D(s)$ must be transformed into a discrete-time difference equation before it can be implemented in the PICs. To make this transformation, the ZOH method is used. It is defined as:

$$H(z) = ZOH(H(s)) = (1 - z^{-1})\mathcal{Z}\left\{\frac{H(s)}{s}\right\} \tag{5.7}$$

where the $\mathcal{Z}$ denotes the z-transform, $H(s)$ is the continuous-time transfer function and $H(z)$ is the discrete-time transfer function. Applying this to the PID controller transfer function $D(s)$ yields the following, where T_s is the sample period:

$$\begin{aligned} ZOH(D(s)) = D(z) &= (1 - z^{-1})\mathcal{Z}\left\{\frac{D(s)}{s}\right\} \Leftrightarrow \\ D(z) &= (1 - z^{-1})\mathcal{Z}\left\{\left(\frac{K}{s} + \frac{K}{T_i s^2} + KT_d\right)\right\} \Leftrightarrow \\ D(z) &= K + \frac{KT_s z^{-1}}{T_i(1 - z^{-1})} + (1 - z^{-1})KT_d \end{aligned} \tag{5.8}$$

To see if this discretization introduces an additional overshoot some simulations with the discrete controller is made. The SIMULINK$^{\text{TM}}$ scheme used in these simulations is shown in figure 5.9.

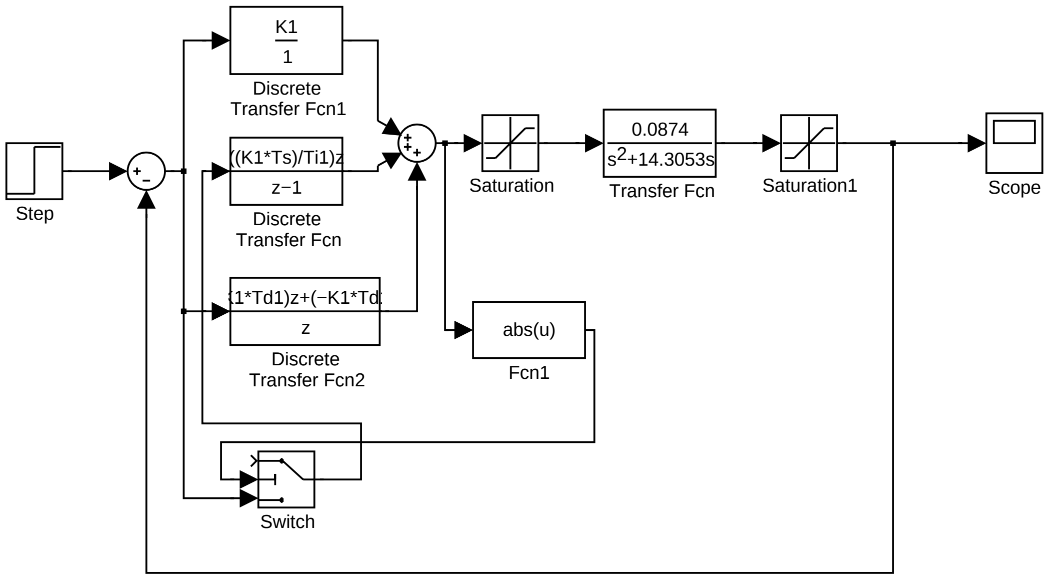


Figure 5.9: The SIMULINK™ scheme used to make simulations with the discrete controller.

In the discrete simulation model integrator anti-windup is included to compensate for the saturations. This is done by disabling the integral part of the controller when the control signal is calculated to be outside the saturation limits. In this case when $|U(z)| > 1024$. This is done in SIMULINK™ by including a switch that disables the integral part of the controller when the control signal is outside ± 1024.

The thing unknown at this moment before the simulations can be made is the sampling frequency f_s. This was in the demand specification set to 125 Hz, but with the slow controller this would cause an oversampling of the system and therefore it is slowed down to 40 Hz.

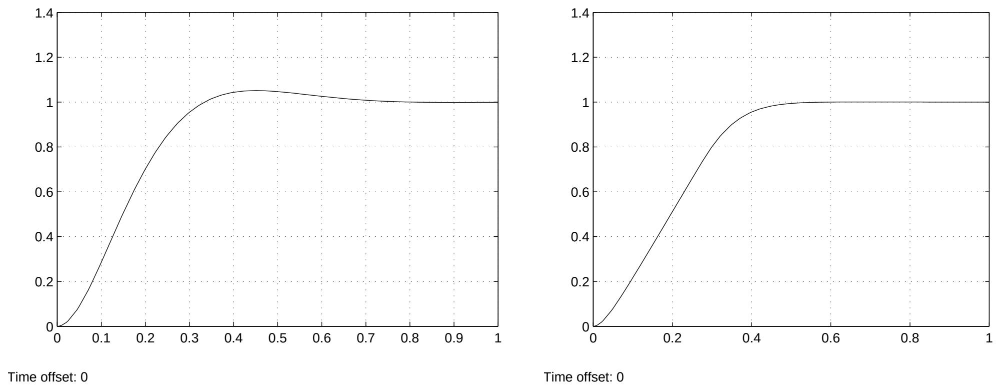
For both type of motors a 1 V step is used to produce a step response from the system. And the results for both motors are shown in figure 5.10.

As seen in the figure the wheel controller system has a considerable overshoot.

In order to reduce this overshoot the wheel controller is detuned. This is done by increasing the differential time. And with the a $T_d = 1$ the system works somehow as wanted. Hereby is meant that there is not an additional overshoot but the rise time demand is not fulfilled by any of the controllers. And with the motors used the rise time demand cannot be fulfilled.

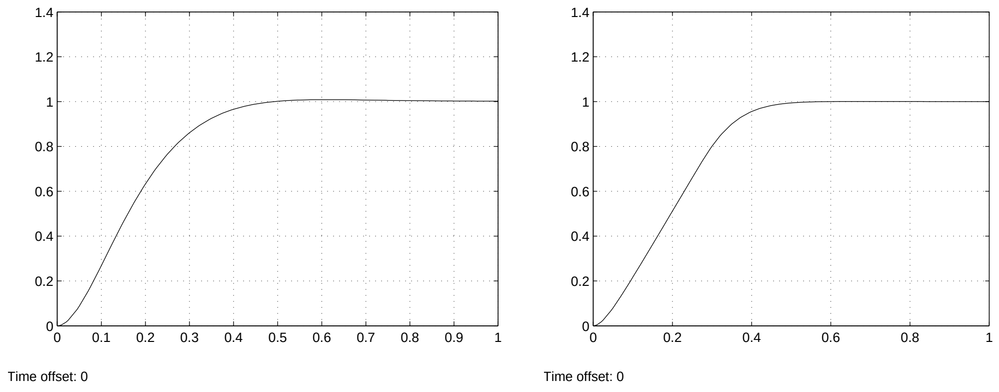
After this detune the step responses in figure 5.11 is obtained.

Before the the controller can be implemented, $D(z)$ from equation 5.8 has to be written as a difference equation.



(a) (b)

Figure 5.10: The step responses of the steering wheel and the wheel motor control system, when the discrete controller is used. (a) Shows the step response for the wheel motors. (b) Shows the step response for the steering wheel motor.



(a) (b)

Figure 5.11: The step responses of the steering wheel and the wheel motor control system, after both controllers are detuned. (a) Shows the step response for the wheel motors. (b) Shows the step response for the steering wheel motor.

Since $D(s)$ describes the transfer function from the error $e(s)$ to the control signal $u(s)$ then $D(z)$ can be rewritten to a difference equation:

$$\begin{aligned} D(z) = & \frac{u(z)}{e(z)} = K + \frac{KT_s z^{-1}}{T_i(1-z^{-1})} + (1-z^{-1})KT_d \Leftrightarrow \\ u(z)(1-z^{-1}) = & \\ & e(z)(K+KT_d) + e(z)\left(K\left(\frac{T_s}{T_i} - 1 - 2T_d\right)\right) z^{-1} \\ & + e(z)KT_s z^{-2} \end{aligned} \tag{5.9}$$

Applying the inverse z-transform ($\mathcal{Z}^{-1}$) to equation 5.9, the following difference equation is obtained:

$$\begin{aligned} u[n] = & \\ & e[n](K+KT_d) + e[n-1]\left(K\left(\frac{T_s}{T_i} - 1 - 2T_d\right)\right) \\ & + e[n-2]KT_d + u[n-1] \end{aligned} \tag{5.10}$$

This equation can be implemented directly in software cause K, T_s, T_i and T_d is known and is listed in table 5.1.

System	K [$\frac{V}{V}$]	T_s [s]	T_i [s]	T_d [s]
Steering wheel	3000	$\frac{1}{40}$	150	0.1
Wheels	1000	$\frac{1}{40}$	100	1

Table 5.1: The PID parameters to be implemented in the PIC software.

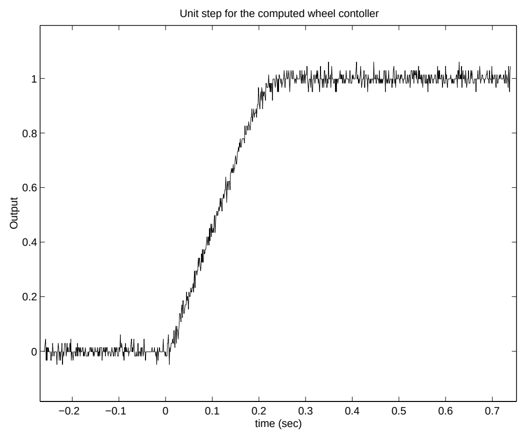
5.4 Test Of Control Algorithm

When the control algorithm for the motors was implemented in software the step responses shown in figure 5.12 was obtained. From this figure it is seen that the motor control system for the steering wheel doesn't act as wanted.

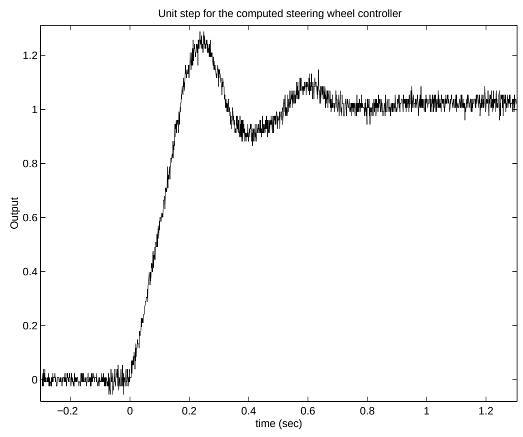
It is assumed that this disagreement between the simulation and the measurements is because of the simple model used for the motors.

This disagreement between the simulated and the measured step responses are caused by the two following simplifications in the motor model used in the simulations:

Non-linearity in the motors: In the modulation of the motors non-linearity was not included in the model. This means that the actual motor does not respond after the same transfer function for changing step sizes in



(a)



(b)

Figure 5.12: The step responses of the two motor systems. (a) show the step response for the wheels. (b) shows the step response of the steering wheel system.

the control signal, as assumed in the model. Which hereby means that the controller designed in this chapter does not act as wanted.

Friction causing a dead zone: When the PWM signal was applied on the motors it was obtained that the motors did not move before the duty cycle became larger than 30%. To compensate for this dead zone an offset of 30% duty cycle is included in the control signal.

Instead of making better models of the motors and after that designing new controllers, some hand tuning of the PID controllers is done.

After this some different values of K_p,T_i and T_d is tried in order to find the best performance of the system.

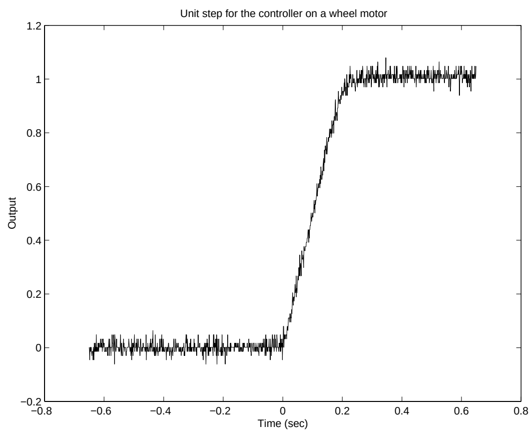
After this hand tuning the step responses of the two types of motors is seen in figure 5.13.

From these measurements it is seen that both controllers fulfill the demand on no overshoot but not the rise time demand.

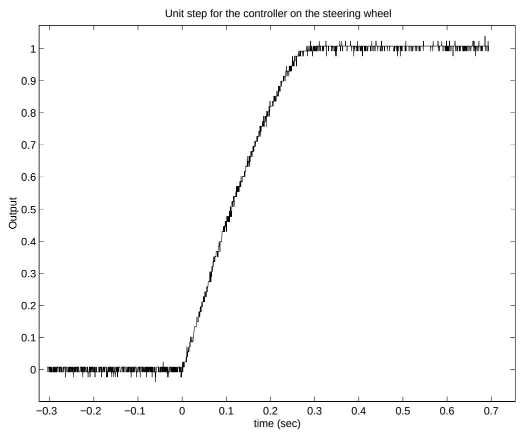
In table 5.2 the PID controller values implemented in the PIC software is shown.

System	K $[\frac{V}{V}]$	T_s [s]	T_i [s]	T_d [s]
Steering wheel	1229	$\frac{1}{40}$	90	2
Wheels	1024	$\frac{1}{40}$	5	1

Table 5.2: The PID parameters implemented in the PIC software.



(a)



(b)

Figure 5.13: The step responses of the two motor systems. (a) show the step response for the wheels. (b) shows the step response of the steering wheel system.

5.5 *Conclusion*

The purpose of this chapter was to make some controllers for the motors used in the power steering system.

First the controllers were designed in continuous time using root loci and step responses.

After that the controller was discretizied and simulated to see if this discretization had caused some additional changes to the control system. Because there actually was included some overshoot in the system two things were done. To compensate for the overshoot caused by the saturation in the control signal an integrator anti-windup was included and to compensate for the remaining overshoot the controller was detuned.

At last the controller was implemented in software and the controllers were tested on the respective motors. And it was concluded that the designed controllers did not work without being changed. This is mainly because of the simple model used in the design, e.g. friction and non-linearity was not included in the model. But after some hand tuning it was possible to make them move some how as wanted.

So all in all, if the control system has to fulfill the rise time demand, some other and faster motors have to be used.

Chapter 6

Software Description

6.1 Initialization

When the three nodes are reset either of the wheel nodes (WNs) measure their current wheel positions and send them to the steering wheel node (SWN). Hereafter, they start running their main loop, using the just sampled value as reference value and wait until they receive the first reference values from the SWN.

Equivalently, the SWN goes in a busy-waiting loop until it has received the current value from both WNs. Then it utilizes its main loop where it calculate the reference values for the two WNs and for itself and sends the two WN values on the CAN bus.

In addition to the procedure mentioned above, the initialization also sets up various registers, timers and directions of I/O ports. This is all dons prior to the routines mentioned above.

6.2 Main Program

The key idea in the fault tolerant nodes is that the two PICs handshake and synchronize their data at regular intervals. This is done with a call to the **synchronize** functions on the master and the slave PICs. The slave data is then sent to the master where it is compared with the master data. The correct data are then returned to the slave and used in the following computations.

In the beginning of the main program the master and slave handshake to ensure that they both start at approximately the same time. Then they wait for a main loop timer (MLT) that triggers the main loop with the sample frequency, which is chosen to be 40 times per second (see chapter 5).

Then they call the function **getMessage** to check to see if a new CAN message is arrived. If there is some flags are set according to this and the message is validated and data are synchronized.

After that, one control loop iteration is run by calling the function **PID**. This causes the master and slave to read the wheel position and synchronize the measured values before the new control signal is computed and applied to the motor.

Finally, the measured (and synchronized) wheel position is sent to the SWN and main loop is ready to be run again.

All flow charts for the wheel node can be seen in appendix A and the flow charts for the steering wheel node can be seen in appendix B.

6.3 Assumptions

The following can be said in general:

- If an error frame is generated by the CAN module, due to CRC errors or alike, this will lead to a retransmission of the data and the faulty data does not reach the input buffer.
- The input buffer of the CAN module is always overwritten by the newest message.
- Alarm frames are placed in a seperate recieve buffer and generate an interrupt.

In the steering wheel the following filosophi is used in connection with the reception of data values from the wheel nodes: If no data is present use old data, if some new data is present, use the newest data. That gives the following two cases.

- If the either the master or the slave pic of the node has recieved new data from one of the motors, and the other has not, the newest data is used.
- It neither the master nor the slave pic has detected new data fron one of the wheel nodes some old data is used.

6.3.1 SPI frame

The communacation over the SPI bus is byte orientated. In each transfer three bytes are transmitted, containing one flag byte and two data bytes, since data is sent as an integer (see figure 6.1. They are sent MSB first.

6.3.2 Flags and registres

OTG Order To Go, pin that indicates that the main loop can start.

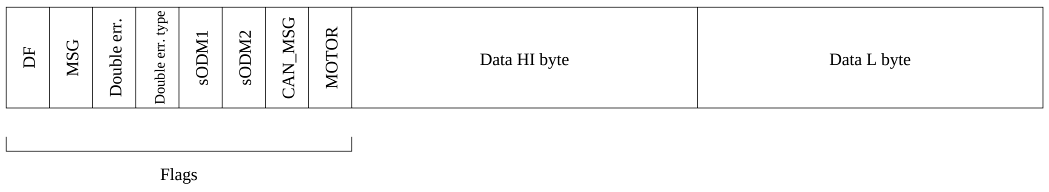


Figure 6.1: The SPI frame.

GO GO, pin that triggers the OTG on the other PIC.

MSG Local flag wich is used by the master (of the nodes) in the message reception routine to determine if a messaage is recieved or the polling schould continiue.

DF Data Flag used to indicate, over the SPI bus, if either the master or slave has send a frame containing data.

Double err. Flag indicating if a double error has ocoured.

Double err. type Flag that indicates the type of double error. If 0 it is an ADC error, else it is a CAN message error.

MCREQ Motor Control Request. Pin on the master to initiate an interrupt on the master to hand over the control of the motor.

RDO Redy Out. Pin used to indicate that either the master or slave is redy to send over the SPI line.

RDI Redy IN. Pin used to indicate that either the master or slave is redy to rescieve data over the SPI line.

MTO Master Time Out. Counter variable to indidate that the master is stuck.

STO Slave Time Out. Counter variable to indidate that the slave is stuck.

SIP Slave Interupt Pin.

IB Interupt pin on master.

M1C Motor 1 Counter. Register that counts the number of times a message with data from motor 1 is recieved.

M2C Motor 2 Counter. Register that counts the number of times a message with data from motor 2 is recieved.

S_ODM1 Slave Obsolete Data Motor 1. Flag that is set when the slave has not recieved data from motor 1, 5 consecutive times.

S_ODM2 Slave Obsolete Data Motor 2. Flag that is set when the slave has not recieved data from motor 2, 5 consecutive times.

M_ODM1 Master Obsolete Data Motor 1. Flag that is set when the master has not recieved data from motor 1, 5 consecutive times.

M_ODM2 Master Obsolete Data Motor 2. Flag that is set when the master has not recieved data from motor 2, 5 consecutive times.

PD Position Disagreement, flag used in the steering wheel node to indicate if the position of the wheels and steering wheel is out of order!

PDC Position Disagreement Counter, register used to count the consecutive number of times that the wheel and steering wheel position is out of order.

VLTOC Very Long Time Out Counter, used while pic's are waiting on the other to send a OTG.

CAN_Count Counter variable to count the number of CAN double errors.

ADC_Count Counter variable to count the number of ADC double errors.

MLT Main Loop Timer, a tiner that makes the main loop run with 40 Hz.

MOTOR Flag that indicates if the data is from motor 1 or 2.

CAN_MSG Flag used locally on the master to indicate is a CAN message is recieved.

resetFlag Local flag used to indicate if the PIC has be reset, and if it has parsed the GO/OTG handshake.

resetTime Counter value that indicates the time the PIC who has reset the other PIC, expects the reset takes.

6.4 The General Cases Software Solution

This section describes the different cases the system can handle. Whenever data is mentioned it covers both data received from the CAN bus and data retrieved from the A/D converter.

Case 1: Either master or slave receives a CAN-messages with corrupted data

The program follow the normal operation until compare. In compare the subroutine rangeCheck is run which finds out if the data is inside valid range. In this case the data from one of the PICs is found invalid. This results in the invalid data to be overwritten by the valid data and both PICs continues with the valid data.

Case 2: Both master and slave receives a CAN-messages with corrupted data

After compare has invoked the rangeCheck on both the slave and master data and found both invalid a double error is signaled. This is done by setting the double error flag in the global flag byte. Further it is indicated through the doubleErrorType flag if the double error is from CAN messages or from ADC data. After this the doubleErrorHandling is run on master. It tests if the doubleError flag is set and if so it first sends the flag byte to the slave and then act according to the type of error. If the doubleErrorType flag is set the error was from CAN messages and the handler requests a retransmission of data. The flag byte is sent to the slave to make sure that the same is sent on both CAN lines.

Case 3: Either master or slave receives a CAN message and the other don't

In the getMessage routine on the wheels the MSG and DF flag is set because when a PIC has received a CAN message. In the compare function these flags are checked. If the MSG is set and the DF is cleared it is tested if the master data is within range. If this is the case the master data is used as syncData and is send to the slave so both PICs use the same data. If the master data is not valid a double error (doubleErrorType=1) is signaled. The same procedure is performed when the slave does not receive data, only difference is that DF is set and MSG is cleared.

The same procedure is performed on the steering wheel with one addition. If the case mentioned above happens five consecutive times the ODM* flag is set in the flag byte. This is used in the compare routine. If it is only the master that has not received data from a motor, the slave data is used as data in both PICs if it is valid. Otherwise a double error is signaled.

Case 4: Steering wheel is disconnected from the CAN bus

The concept is the same as in case 3, but here neither MSG nor DF is set. In this case compare does not change any values and therefor the previous data is used. The case is vaild for both wheel and steering wheel nodes.

Case 5: Either master or slave is stuck somewhere in software

If the slave does not make the handshake, the PICs run normally from the start of their main loops through message reception down to synchronize. Here the slave fails to make the handshake, and the master aborts the handshake operation and goes into a stoOverflowHandling. This handler asserts the SIP pin on the slave and sets the resetTime on the master. Asserting the SIP pin results in an interrupt on the slave that sends a CAN alarm 0 and resets the slave. The alarm is send by both master and slave. After this the synchronize routine continues on the master where it tests if the data on the master is inside the valid range. If so, it returns this data as syncData. If the data is not valid, the last valid data is used as syncData. The master continues to operate until the slave has gone through the reset.

The main part of this exception is the same above. In this case the master misses the handshake and the slave goes into a mtoOverflowHandling. The slave asserts the MCREQ pin on the master and sets the resetTime. Asserting the MCREQ on the master makes the master go into an interrupt service routine that hands over the MUX control to the slave and resets the master. After this the slave continues like the master did in case 1.

Case 6: Either master or slave samples invalid motor position data

In the compare routine the sampled data is run through the rangeCheck. If one set of data is invalid this data will be omitted and the valid data will be used.

If both sets of data is invalid, the larst valid data will be used, and a CAN alarm will be broadcasted.

If both sets of data are valid but not equal the average value is used as data in both PICs.

Case 7: Either master or slave receives a CAN message with wrong data and the other receives nothing

In this case the two PICs reaches the compare function. Here either the MSG or DF flag is cleared, meaning that the data that is present will be run through rangeCheck. Here the data is found invalid which results in a double error. This is send by both master and slave and the two PICs use the larst synchronized valid data.

6.5 Header Description

/* SEE figure B.4: “Control loop”

RUNS ON: master and slave PIC on all three nodes.

PURPOSE: to retrieve A/D controller value, to synchronize the measured value (between master and slave), to calculate the control signal (duty cycle and direction) and to apply this to the motor outputs. */

int PID(int reference) { fla,fla }

/* SEE figure A.2: "ISR for message reception in the wheel"

RUNS ON: the two wheel nodes.

PURPOSE: Uppon CAN interrupt this message will signal that a CAN message is received. Then it will test if the message is an alarm, and call the alarm handler or it is normal data. When normal data is received, this will be stored in the global "tempData" variable and the function returns. */

int CAN_Read (void) { if message is an alarm: call alarmhandling; else store the incomming data as tempData; }

/* SEE figure B.13 "Retrieve A/D converter value"

RUNS ON: master and slave on all three nodes.

PURPOSE: retrieve the A/D converter value from the ADC buffer and store it in a variable. */

int AD_Read(int *result) { start the ADC; wait until the conversion is done; store the value; }

/* SEE figure B.5: "Synchronize data"

RUNS ON: the master on all three nodes (the code differ from SWN to WN, but the functionality is the same).

PURPOSE: to handshake with the slave and exchange data so that both master and slave PIC continuous from here with the same data values.

First the function sets the global variable "tempData" to -1 (this is possible since the value of "tempData" is known localy in "synchronize" as data). This is done to indicate that this message has been used. So if "tempData" is -1 befor the next call of "synchronize" no new messages has arrived and the corresponding flags will not be set (handlet in message reseption).

Before the handshake (RDI/RDO) is evoked, the function tests if the other PIC on the node is beeing reset. This is done with the "resetFlag" and "reserTime". If these are set, there is no reason to perform the handshake with the other PIC. In this case the handshake is omitted and the function only tests if the data is indside valid range. Otherwise the two PICs handshake, exthange data and the master desides what data to use.

*/

int synchronize(int data, int *syncData) { if (resetTime & resetFlag is set) dont handshake, just rangeCheck own data; return; else handshake with slave; get “tempData” from slave and store this as “sData”; call function “compare”; send “syncData” to slave save “syncData” as “previousSyncData” return; }

/* SEE figure B.5: “Synchronize data”

RUNS ON: the slave on all three nodes.

PURPOSE: to handshake with the master and exchange the correct data from the function call. As with the function on the master resetTime and resetFlag is tested.

The same actions are taken here as in the master with regards to the resetTime and resetFlag. If non of them is set, the “sData” argument is sent to the master and the correct, synchronized data are returned as “syncData”. Further a test is performed to se if the master has detected a double error. */

int synchronize(int sdata, int *syncData) { if (resetTime & resetFlag is set) dont handshake, just rangeCheck own data; return; else handshake with master send “sData” to master call function doubleErrorTest; save returned value as “previousSyncData” return; }

/* SEE figure A.6: “Compare received data”

RUNS ON: the master on the two wheel nodes.

PURPOSE: to compare the arguments “mData” and “sData” and determine waether they are the same. If not the right actions must be taken so that both PICs continous with the same data. */

int compare(int mData, int sData, int *SyncData, int *previousSyncData) { fla, fla }

/* SEE figure B.7: “Compare received data”

RUNS ON: the master on the steering wheel node (same purpose but code is different).

PURPOSE: to compare the arguments “mData” and “sData” and determine waether they are the same. If not the right actions must be taken so that both PICs continous with the same data. */

int compare(int mData, int sData, int *syncData, int *previousSyncData) { fla, fla }

/* SEE figure B.8: “Double error handling (runs on master)”

RUNS ON: the master on all three nodes.

PURPOSE: to signal to both the slave PIC and the other nodes, that an error has occured and what actions they have to take in that context. The signal to the slave is given through the SPI byte and all other nodes receive an alarm frame on the CAN bus

The contents of the alarm frame depends on the “doubleErrorType” flag in the global flag byte. In case of erroneously received CAN messages the alarm frame should invoke a retransmission. In case of wrong A/D converter values the alarm frame indicates this. If the same error has occured ten consecutive times the function signals that the system is dooemd! */

int doubleErrorHandler(unsigned char *canCount, unsigned char *adcCount) { set doubleErrorFlag = 1) if (doubleErrorType = 1) signal CAN error to slave and send CAN message; if (doubleErrorType = 0) signal ADC error to slave and send CAN message; if (oneError = 10) signal Doomed on CAN; }

/* SEE figure B.6: “Double error test (runs on slave)”

RUNS ON: the slave on all three nodes.

PURPOSE: to check if the SPI data returned from the master indicate a double error. If so send a CAN message coresponding to the type of error indicated by the doubleErrorType flag in the SPI byte. After 10 consecutive double errors of one type it sends out an alarm frame signaling that the node is doomed. */

int doubleErrorTest(void) { if (doubleErrorFlag == 1) { if (doubleErrorType = 1) wrong CAN data received send retransmission request; else wrong ADC measurements send alarm frame indicating ADC error; } if (oneError = 10) send alarm frame signaling that the node is doomed }

/* SEE figure B.11: “MTO overflow error (runs on slave)”

RUNS ON: the slave on all three nodes.

PURPOSE: to get the MUX control from master and reset the master if it does not make the RDI/RDO handshake in the synchronize function.

If the master does not respond in the RDI/RDO handshaking in the synchronize function, take over the MUX control, set the resetTime, send a CAN alarm and reset the master. */

int mtoOverflowHandling(unsigned char *resetTime) { set resetTime = something; toggle MCREQ on master PIC to invoke an ISR that hands over motor control; send CAN alarm; return; }

/* SEE figure B.9: “Interrupt subroutine for MCREQ (runs on master)”

RUNS ON: the master on all three nodes.

PURPOSE: to hand over the MUX control to the slave.

If the slave toggles MCREQ as a result of missing response in the RDI/RDO handshak in the synchronize function, hand over the motor control, send an alarm frame and reset.

*/

int ISR_timeout(void) { negate MUX pin to hand over motor control to the slave; send CAN alarm; reset; }

/* SEE figur B.12: "STO handling (running on master)"

RUNS ON: the master on all three nodes.

PURPOSE: to be able to reset the slave in case of missine RDI/RDO handshake in the synchronize function.

If the slave didn't respond in the handshake, set the resetTime, send an alarm frame and invoke an ISR that resets the slave. */

int stoOverflowHandling(unsigned char *resetTime) { set resetTime = something; toggle SIP; send CAN alarm; return; }

/* SEE figure B.10: "Interrupt subroutine for STO interrupt (runs on slave)"

RUNS ON: the slave on all three nodes.

PURPOSE: to reset the slave if it misses the RDI/RDO handshake in the synchronize function.

If the master toggles SIP as a result of missing response in the handshaking, send an alarm frame and reset the slave. */

int ISR_timeout(void) { send CAN alarm; reset; }

/* SEE figure B.2: "compute motor positions"

RUNS ON: the master and slave on the steering wheel node.

PURPOSE: to calculate new reference values to the three motors and store these in global variables. Make sure that the results fits the main protocol. */

int motorPositions(void) { fla, fla }

/* SEE figure B.14: "ISR for CAN message reception"

RUNS ON: the master an slave on the steering wheel node.

PURPOSE: to receive and sort CAN messages from the two wheel nodes. Saves messages to tempMot1 and tempMot2. In case of an alerm frame, react acording to the context of the alarm. In case of obsolete data from either motor, the master sets the mODM* flag, which is a gobal variable on the master. If the slave detects obsolete it signals this by setting the ODM* flag in the SPI byte. This secures that both PICs opereta on the newest data if one PIC has received new data. */

void Can_Read(void) { if (message == Alarm) call alarmhandling; if (message == motor1) save message as tempMot1; if (message == motor2) save message as tempMot2; if (olddata == something) set appropriate ODM flag; }

/*

SEE figures A.13, A.14, and B.15: “Alarm handling for wheel 1”, “Alarm handling for wheel 2” and “Alarmhandling for ste steering wheel”.

RUNS ON: All nodes and all PICs as an ISR when an alarmframe is recieved. (The code is different from node to node, but the purpose is the same all over!)

PURPOSE: to handle the different alarm frames send on the CAN bus.

The different types of alarms are signaled by transmitting different codes in the alarm message. It is able to evoke a retransmission of data if such is requested and dump the other alarms! */

int alarmHandling(int alarmFlags) { if (alarmMessage == something) do something; else don’t care!!; }

/*

SEE figure B.3: “Message recption”

RUNS ON: Steering wheel node! (the code does not differ in the master and slave apart from the MSG and DF flags)

PURPOSE: take the different messages recieved from the two wheels and compare them, so the master and slave PIC has the same set of data to work on!

First it is tested if the “tempMot” variable is -1. If it is, no new CAN messages has arrived from this wheel and therefor neither the CAN_MSG nor the MSG/DF flag must be set. This test is performed on the data from both wheels.

*/ int msgReception(void) { if (tempMot1 != -1) set flags; call synchronizs on tempMot1; store syncData as curValMot1; if (tempMot2 !=-1) set flags; call synchronizs on tempMot2; store syncData as curValMot2; clear flags; }

/*

SEE figure: Not present.

RUNS ON: Master and slave of the three nodes.

PURPOSE: to send the “input” argument together with the global SPI byte to the slave.

*/ int SPI_Write(int input) { split int input up into two bytes; send SPI byte and wait untill bus is ready; send first data byte and wait; send larst data byte and return; }

/*

SEE FIGURE: Not present.

RUNS ON: Master and slave of all three nodes (The code is not the same, but the functionality is the same. The differense is which flags in the global SPI byte the function is allowed to alter)

PURPOSE: To receive the data on the SPI bus and place it in the right places.

*/ int SPI_Read(int *input) { when the receive buffer is full empty it and store the value in a char array; when the char array is full, put the first value in the SPI byte and collect the larts two values to output; }

6.5.1 Sub Routine Headers

/* SEE figures B.5, B.7 and A.6.

RUNS ON: All nodes, it is a common function overall.

PURPOSE: to determine if the input data is inside the alowed range! This is signaled using the return values of the function. */

int rangeCheck(int currentValue, int previousValue) { if (currentValue indside range) return 0; else return something else!; }

/* SEE figurs B.7 and A.6.

RUNS ON: Part of the compare functions that runs on all masters.

PURPOSE: to determine if the type of double error made in the compare is due to a CAN message or an ADC error. If the error is due to a CAN message, indicate this by setting the double error flag.

int determineDoubleErrorType(void) { if (CAN_MSG == 1) set doubleErrorType = 1; else set doubleErrorType = 0; }

/*

SEE FIGURE: Not present.

RUNS ON: All PICs as a part of the init function.

PURPOSE: To make sure that the two PICs of a node has reached the same place in the init function.

*/ void handshakeing(void) { set RDO; when RDI = 1; clear RDO and return; }

/*

SEE FIGURE: A.15 and B.16

RUNS ON: All PICs

PURPOSE: makes sure that the main loop is run at a frequency of 40 Hz and tryes to handshake with the other PIC of the node. Is this not possible, then this PIC continous in its main loop after VLTOC has made overflow.

In order to make sure that the two PICs start their timers simultaneously (which menas thet the two PICs use as little time as possible to wait on eachother) the "gotimer()" tests if it is the first time it is run. This is done with the "init" variable. On the first call of "gotimer()" "init" will be set to 1. This means that the VLTOC is bypassed, and the PIC will wait until the OTG is set from the other PIC. This also means that if the other PIC never sends the OTG, the node will never start. On the other hand, when the OTG arrives, the "init" variable is set to 0 and the VLTOC is used on the next call of "gotimer()".

*/ void gotimer(void) { wait until 40 Hz timer has run out; set gopin = 1; wait until OTG = 1 or VLTOC overflow; if (init = 1) wait only on OTG; set init = 0; set gopin = 0; reload timer; }

/* SEE FIGURE: Not present.

RUNS ON: The steering wheel node as a subfunction in motorPositions().

PURPOSE: to determine if the to arguments "pos1" and "pos2" are within a suitable range.

Ths function is used to introduce a tolerance in "motorPositions()", in order to ensure that if the two arguments are within the margin, they are treatet as equal. If they are within the margin, the function returns 0, else it retuens 1. */

void comparePositions(int pos1, int pos2) { if ((pos1 - pos2) is within MAXDIF) return 0; else return 1; }

6.5.2 Init Functions

To setup all the different functionallyties used in the functions discriped above, a list of init functions is used. The main init function (den er beskrevet andensteds, men skal opdateres!) is:

void Init(void)

Apart from this the other functions are the following, where their name tells what they initialize;

void PWM_Init(void)

int Can_Init(void);

void AD_Init(void);

void SPI_Init(void);

void Port_Init(void);

void Interrupt_Init(void);

6.5.3 Other Functions

int PWM_Write(int dutycycle); int AD_Read(int* result); void Can_Read(void); int Can_Write(int Id, int msg);

6.6 Software Functions

This section tells how the microcontroller is set up for use in this application and afterwards some simple functions are described.

6.6.1 PIC setup

The PIC18F458 has a lot of built in features such as timers and serial interfaces. To use those they has to be set up. This is done in a few initialization functions which will be described below.

A/D converter. The PIC has a 10 bit A/D converter with 8 possible inputs. Input pin2 (AN0) is used for the conversion. A stable reference voltage for the A/D converter is made by an 78L05 voltage regulator. It's connected to the PIC on pin 4 and 5 (V_{REF-} and V_{REF+}). Only the 8 of the 10 bits are used for operation, because of noise on the two least significant bits.

PWM signal. To make the PWM signal the built in Enhanced/Capture/ Compare/PWM module is used. The PWM register is 10 bit. The frequency is chosen to be 1.22 kHz beacuase of some limitations in the motor drivers. The frequency is set up by timer2. Output pin 27 is assigned to be the PWM output pin.

The direction of the motor is decided by pin 39 and 40 (PortB.6 and PortB.7) , which is set as output pins.

CAN Bus setup. The PIC is implemented with a full CAN system and it supports all protocols up to the CAN2.0B Active and Passive protocol. It contains two input buffers but it's not necesary to use more than one, so only buffer 0 is used for this application. The CAN Bus initialization function sets up the CAN masks and filters for input buffer 0 and the recieve buffer 0 interrupt is enabled.

The ID-number for transmission is also set in the initialization function. The baudrate is set to 625 Kb without argumentation. Pin 35 and 36 is assigned to CANTX and CANRX and they are connected to the CAN-driver LM119.

6.6.2 Functions

`int AD_Read(float* result)`. The `AD_Read` function starts the A/D converter and samples the input pin specified in the initialization. The 8 MSB are multiplied with 0.0195 and stored in a float variable `result` that will the contain the voltage between 0 and 5 V. The microcontroller will wait for the A/D converter to finish converting before the function ends. This will take approximate 16 μs.

`int PWM_Write(float dutycycle)`. The `PWM_Write` function writes the specified dutycycle yo the output pin as sepcified in the initialization. The dutycycle parameter `dutycycle` must be a value between -5 and 5, corresponding to the output voltage to the motor. Values below 0 V will make the motor drive in one direction and values above 0 V will make the motor drive the other direction. The `dutycycle` is multiplied with 204.8 in order to make it a 10 bit value for the PWM control register and written to the PWM output pin.

The direction is written to the direction output pins as described in the initialization.

`int Can_Read(int *id, unsigned char *length, unsigned char *msg)`. The `Can_Read` function reads the recieve buffer and puts the identifier of the message in `id`, the lenght of the message in `length` and the message in `msg`. It also cleares the RXFUL bit, to make the buffer ready to recieve a new message.

`int Can_Write(int Id, unsigned char length, unsigned char *msg)`. The `Can_Write` function writes the `msg` to the CAN bus and it uses the `Id` as its transimssion identifier. If the `length` is zero it sets the transimssion frame remote transmission request bit.

`int PID(float reference)`. The `PID` function is the controller function. The controller equations and parametres is in this function and it used the `AD_Read` and the `PWM_Write` functions to sample and write to the motor. It also contains integrator wind up for the controllers.

Chapter 7

Test Specification

The tests for the system is based on the General Cases described in chapter 3.2.

The tests are made while the system is running under normal conditions. To start the system both wheels and steering wheel must point in approximately the same direction before the power is turned on. Under all tests the data on both CAN lines are logged by the CAN bus sniffer, and saved to the disk. To plot the results a MATLAB^{TM} program is create to each test.

Test 1: Either master or slave receives a CAN-messages with corrupted data

The test is performed to ensure that the system will continue working if master or slave on a node becomes a "babbling idiot". I.e. wrong data is transmitted from a node on either master or slave bus. The test is performed twice, one with a slave transmitting wrong data and one with a master transmitting wrong data.

The first test is performed by forcing the master on the steering wheel to send out wrong data. This is done in the main loop of the steering wheel software, where the steering wheel is forced to send out the wrong value of 2000 instead of the right value. To be able to change the reference while the system is running it is activated by polling on a I/O pin. When the pin becomes high the right data is changed to wrong data. While the wrong data is send the system should be working as normal.

A similar test is performed where the slave on the steering wheel sends the wrong data.

Test 2: Both master and slave receives a CAN-messages with corrupted data

A test where both master and slave on the steering wheel sends wrong data is performed. In this case both wheel nodes should use the last correct reference position from the steering wheel. The wheel nodes should also transmit an alarm (CAN error message 32) to indicate a that they receive wrong data on both CAN lines.

The test is performed the same way as test 1 where both master and slave sends out wrong data.

Test 3: Either master or slave receives a CAN message and the other don't

This test is made to ensure that the system will continue working when one CAN line is broken.

The test is made two times, one where the master CAN line is disconnected and one where the slave CAN line is disconnected. The first test is performed by disconnecting the CAN line to the master of a wheel node. The CAN line is disconnected while the system is running. No interruptions on the system should be seen.

The logged data is plotted in MATLAB™ and from the graphs it can be verified that the system continues to work when data from one can line is missing.

Test 4: Steering wheel is disconnected from the CAN bus

This test made to ensure that the wheel nodes will use the last correct reference when both CAN lines from the steering wheel is disconnected.

To be able to log the data on the CAN lines this test is done by forcing the steering wheel to stop sending data. This will simulate that both can lines are disconnected. The data from the steering wheel is deactivated while the system is running. This is done by polling on a I/O pin in the main loop of the steering wheel. If the polled I/O pin is high no data is send from the steering wheel.

The logged data is plotted in MATLAB™ and it should verify that the wheel nodes uses the last known correct reference. And from the log file it should be verified that a CAN alarm (CAN error message 32) indicating that wrong data is received on both CAN lines.

Test 5: Either master or slave is stuck somewhere in software

This test is made to ensure that the system will continue working after either master or slave dies on a node. And test that the working PIC will reset the one that died.

The test is performed by forcing the master of a wheel node into a infinite loop to simulate that the PIC died. To enter the while(1); loop an I/O pin is polled at the beginning of the main loop and the infinite loop is entered if the I/O pin is high. The pin is polled high while the system is running. A similar test is performed the slave on a wheel node dies.

The logged data is plotted in MATLAB^{TM} and it should be verified that the PIC that dies stops sending data, but the system is still working. After the PIC who died has been reset it should start sending data again.

Test 6: Either master or slave samples invalid motor position data

This test is made to ensure that the system will continue working if a wire to one AD converter is disconnected.

The test is made by disconnecting the wire to the AD converter on the master on a wheel node. A similar test is performed by disconnecting the AD converter to the slave on a wheel node.

The logged data is plotted in MATLAB^{TM} and it should be verified that the system will continue working.

Test 7: Either master or slave receives a CAN message with wrong data and the other receives nothing

The test is made to ensure that the wheel nodes remain at the same position when receiving wrong data or no data from the steering wheel.

The test is performed by making the master on the steering wheel send out wrong data and disconnecting the CAN line from the slave. The wrong data is made as in test 1. The similar test is made where the slave on the steering wheel send out wrong data and the CAN line from the master is disconnected.

The logged data is plotted in MATLAB^{TM} and should verify that the wheel nodes uses the last known correct reference. And from the log file it should be verified that a CAN alarm (CAN error message 32) indicating that wrong data is received on both CAN lines.

Chapter 8

Test results

The results are made out from log-files at the CD-ROM.

8.0.3 Test 1

The test were performed to ensure that the system will continue working if master or slave on a node becomes a "babbling idiot".

The result for the test with the master sending corrupted data is plotted at figure 8.1.

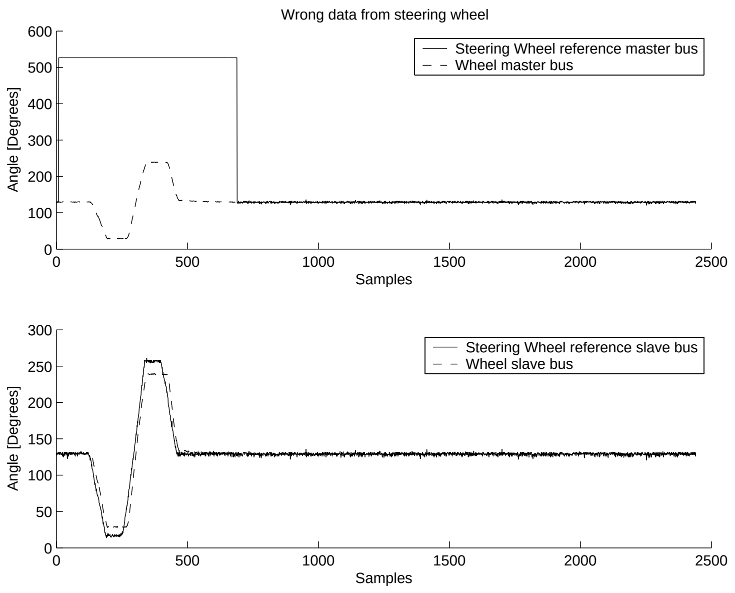


Figure 8.1: The two graphs shows the reference from the steering wheel to the node and the motor position from the node at the two CAN bus lines(master at the top graph and slave at the bottom graph).

When the corrupted data(value 526) occurs the wheel do not chance its position. The node uses the reference value from the slave bus, see lower graph at figure 8.1. According to Case 1 in chapter 3.2 the result is correct.

The result for the test with the slave sending corrupted data is plotted at figure 8.2.

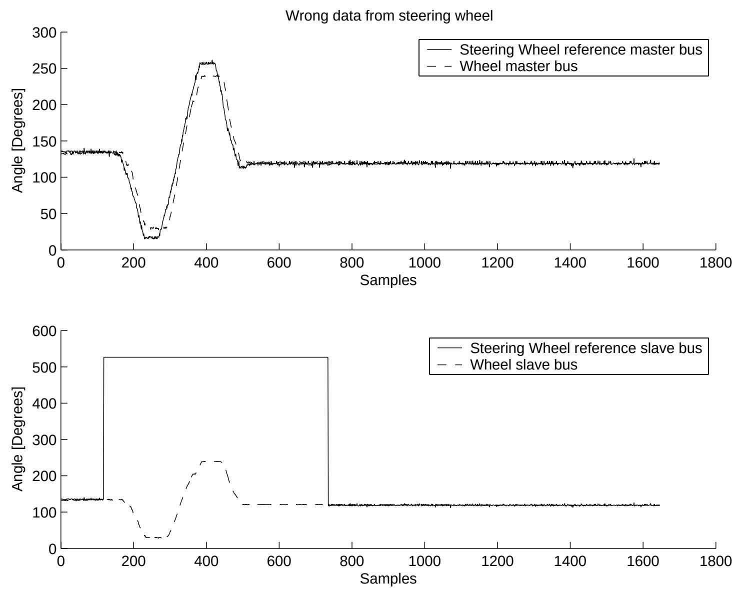


Figure 8.2: The two graphs shows the reference from the steering wheel to the node and the motor position from the node at the two CAN bus lines(master at the top graph and slave at the bottom graph).

The result is the same as the result from the master.

8.0.4 *Test 2*

The test with both PICs in the wheel node receiving corrupted data is plotted at figure 8.3

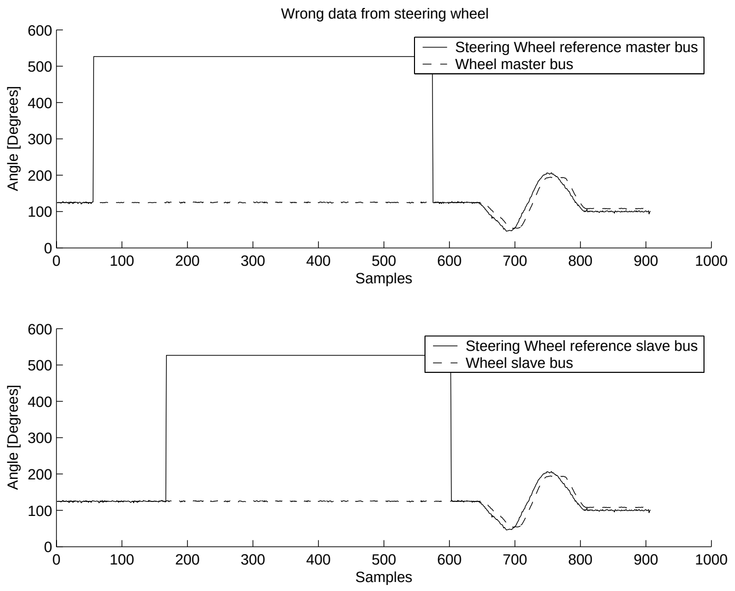


Figure 8.3: The two graphs shows the reference from the steering wheel to the node and the motor position from the node at the two CAN bus lines(master at the top graph and slave at the bottom graph).

The graphs shows that with corrupted data(value 526) at both CAN-bus lines the node uses the last correct reference value received from the steering wheel. According to Case 2 in chapter 3.2 the result is correct.

8.0.5 *Test 3*

This test was made to ensure that the system would continue working when one CAN line was broken.

The result for the test with the master CAN line disconnected are plotted at figure 8.4.

At the top graph it can be seen that the master CAN bus line are disconnect at a sample value just over 300. The CAN line are connected again approximately at 950. Under the disconnection the motor position follows the references received by the slave as shown at lower graph. According to Case 3 in chapter 3.2 the result is correct.

The result for the test with the slave CAN line disconnected are plotted at figure 8.5.

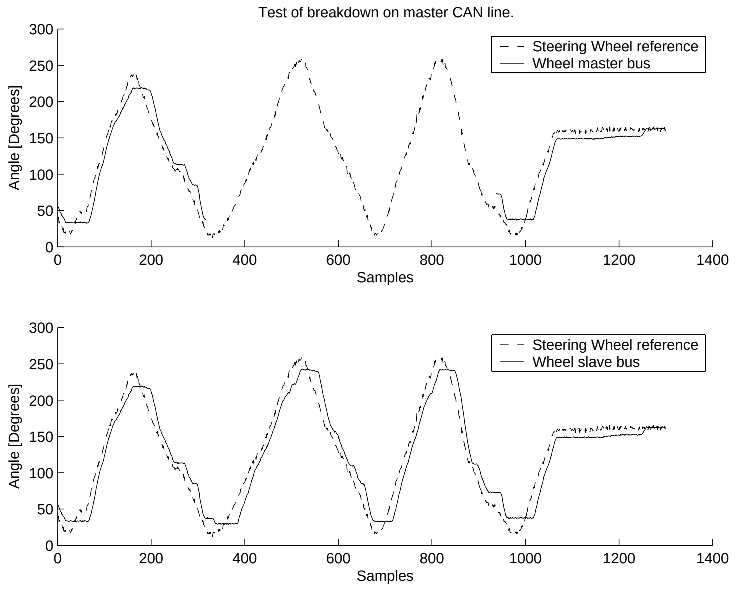


Figure 8.4: The two graphs shows the reference from the steering wheel to the node and the motor position from the node at the two CAN bus lines(master at the top graph and slave at the bottom graph).

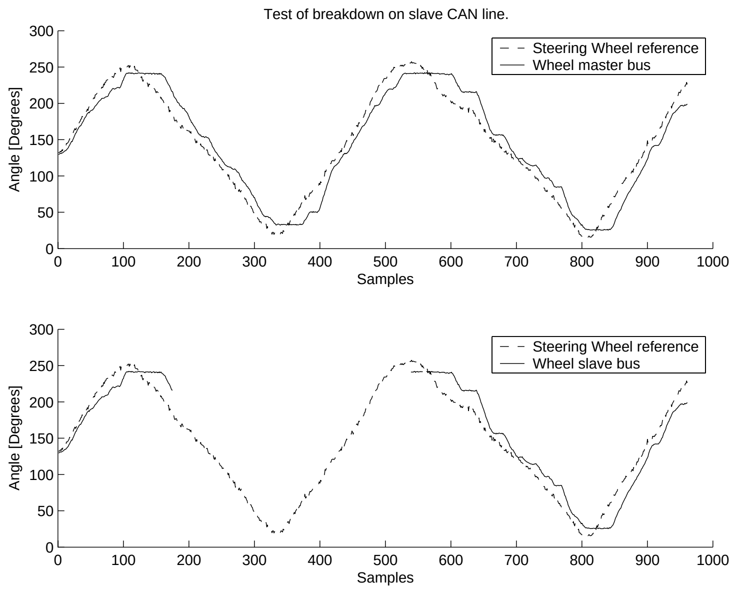


Figure 8.5: The two graphs shows the reference from the steering wheel to the node and the motor position from the node at the two CAN bus lines(master at the top graph and slave at the bottom graph).

The result is the same as the result from the master.

8.0.6 Test 4

This test were made to ensure that the wheel nodes would use the last correct reference when both CAN lines from the steering wheel were disconnected.

The result from the test are plotted at figure 8.6

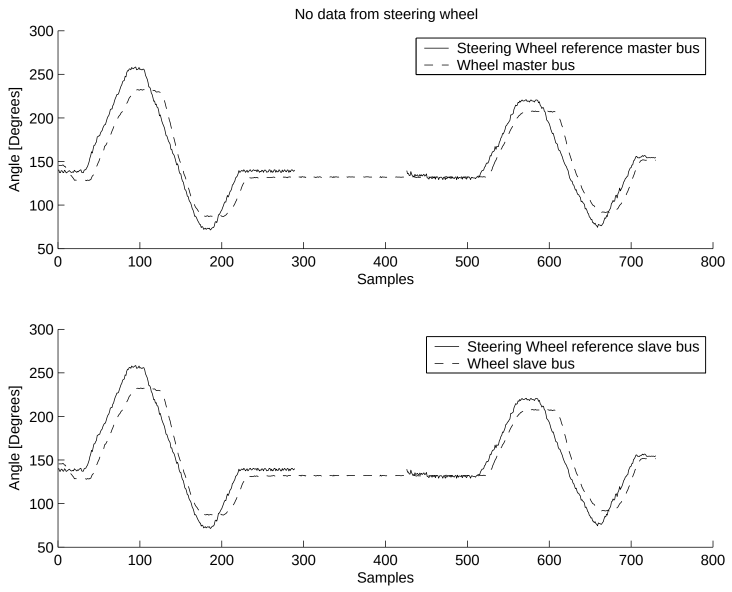


Figure 8.6: The two graphs shows the reference from the steering wheel to the node and the motor position from the node at the two CAN bus lines(master at the top graph and slave at the bottom graph).

As figure 8.6 shows that both CAN lines are missing reference data from the steering wheel at samples from 300 to 400. And the node uses the last reference during the period. According to Case 4 in chapter 3.2 the result is correct.

8.0.7 Test 5

This test is made to ensure that the system will continue working after either master or slave dies on a node. And test that the working PIC will reset the one that died.

The results for the test with the master dead are plotted at figure 8.7.

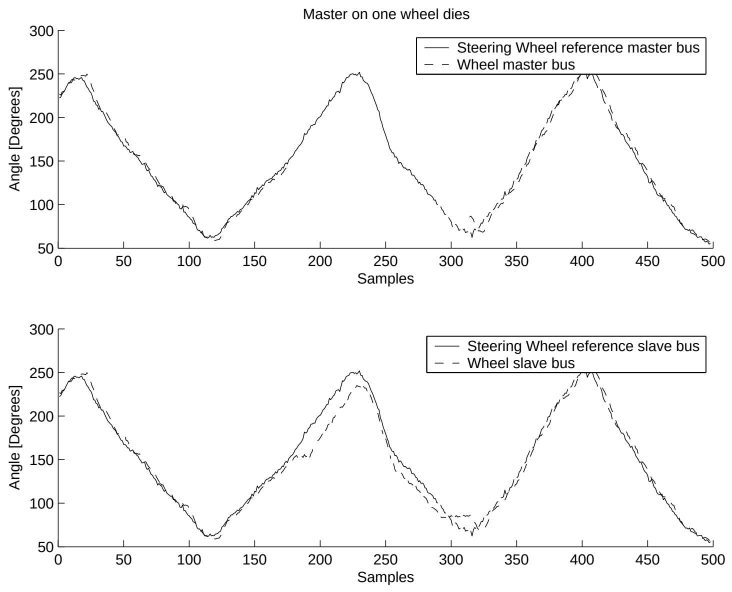


Figure 8.7: The two graphs shows the reference from the steering wheel to the node and the motor position from the node at the two CAN bus lines(master at the top and slave at the bottom).

The figure show that master on a wheel stop sending data for a periode and then recovers. This could indicate that it has been reset by the slave as it should be. While the master is down the slave continues working and follows the reference from the steering wheel. According to Case 5 in chapter 3.2 the result is correct.

The result is similar where the slave dies and is plotted in figure 8.8.

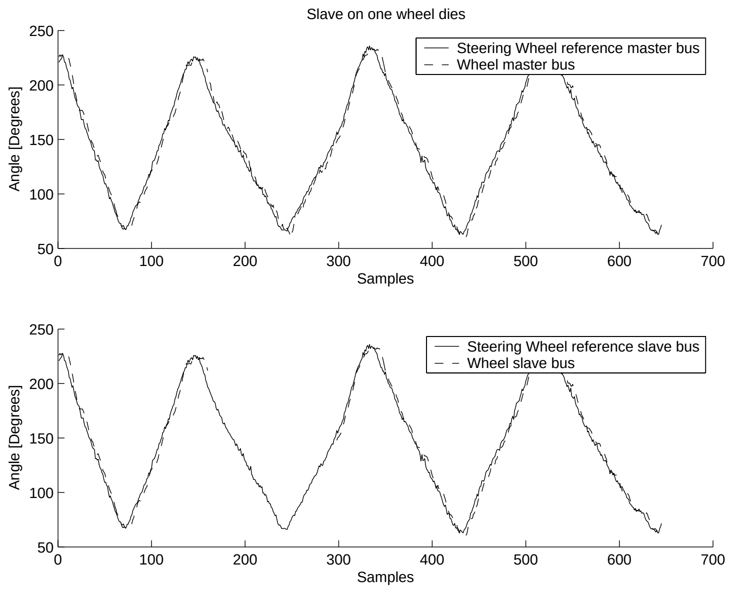


Figure 8.8: The two graphs shows the reference from the steering wheel to the node and the motor position from the node at the two CAN bus lines(master at the top and slave at the bottom).

8.0.8 Test 6

This test was made to ensure that the system would continue working if a wire to one AD converter was disconnected.

The result for the test with the wire to the AD converter at the master PIC disconnected are plotted at figure 8.9.

The two graphs shows that the disconnected AD converter causes no error to the system. According to Case 7 in chapter 3.2 the result is correct.

The result for the test with the wire to the AD converter at the slave PIC disconnected are plotted at figure 8.10.

The result is the same as the result from the master.

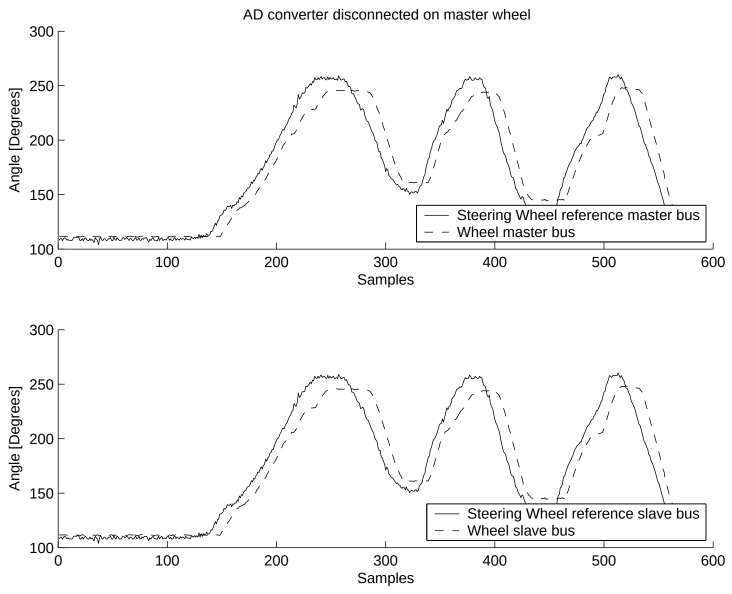


Figure 8.9: The two graphs shows the reference from the steering wheel to the node and the motor position from the node at the two CAN bus lines(master at the top and slave at the bottom).

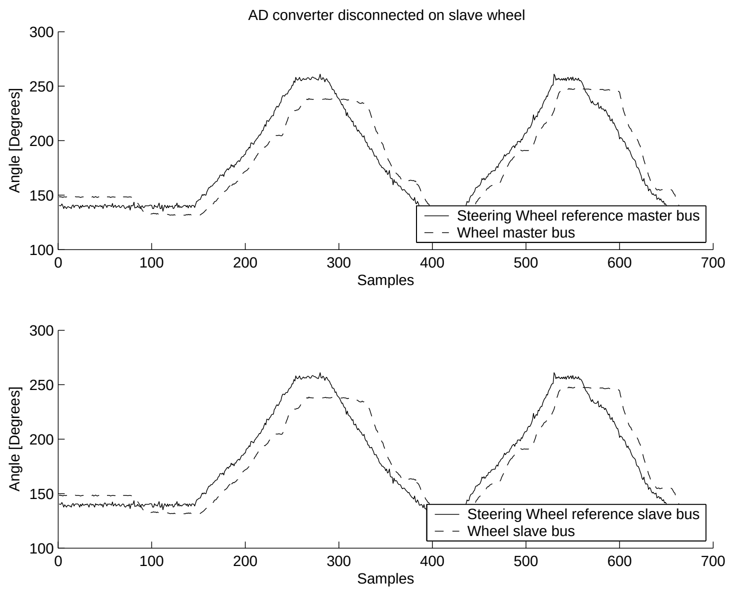


Figure 8.10: The two graphs shows the reference from the steering wheel to the node and the motor position from the node at the two CAN bus lines(master at the top and slave at the bottom).

8.0.9 Test 7

The test is made to ensure that the wheel nodes remain at the same position when receiving wrong data or on data from the steering wheel.

The result for the test with the master CAN line disconnected and corrupted data at slave CAN line are plotted at figure 8.12.

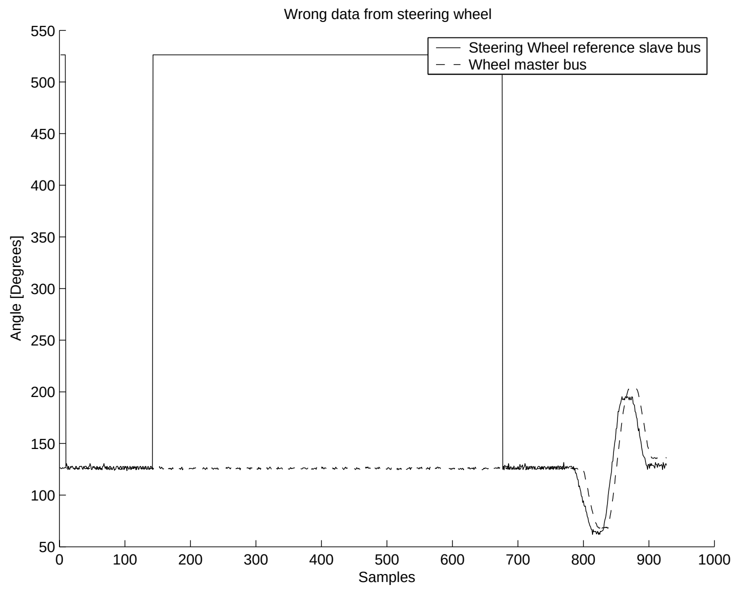


Figure 8.11: The graph show the reference from the steering wheel to the node and the motor position from the node.

At the graph it can be seen that corrupted data are send from the steering wheel from around 150 to around 675 samples. The node holds its motor position at the last correct received reference. When correct data from the steering wheel are sent again the wheel follows. According to Case 3 in chapter 3.2 the result is correct.

The result for the test with the slave CAN line disconnected and corrupted data at slave CAN line are plotted at figure 8.11.

The result is the same as the result from the master.

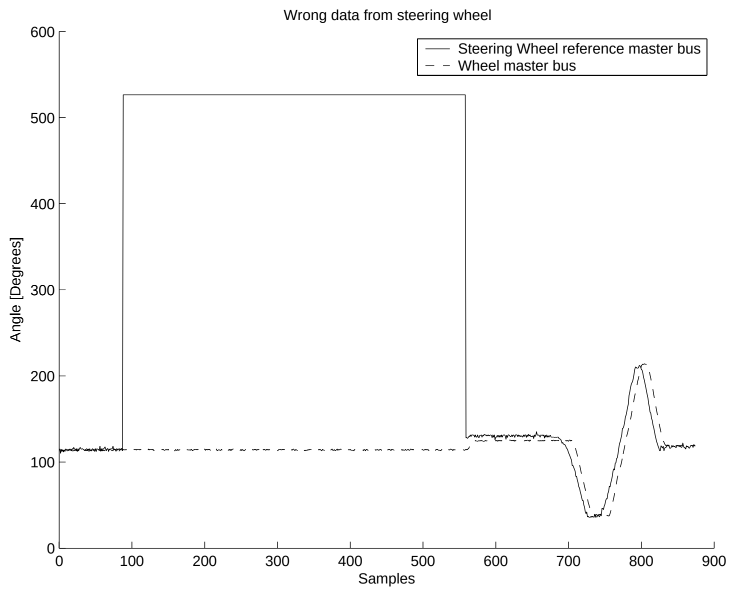


Figure 8.12: The graph show the reference from the steering wheel to the node and the motor position from the node.

Bibliography

[cdrom] Enclosed CDROM for this project

[dc] Franklin, Powell & Workman *Digtal Control of Feedback System* Addison Wesley ISBN: 0201820544

[fc] Franklin, Powell & Emami-Naeini *Feedback Control of Dynamic System* Addison Wesley ISBN: 0201527472

[fdm] FDM *Alt om bilen* Det Bedste fra Reader's Digest A/S, København 1974.

[PICdata] Microchip Technology Inc. *PIC18FXX8 Data Sheet* U.S.A., Microchip Technology Inc., 2002.

[PICpro] Microchip Technology Inc. *PIC18FXX2/FXX8 Programming Specifications* U.S.A., Microchip Technology Inc., 2001.

[CAN] Robert Bosch GmbH *CAN Specification Version 2.0* Stuttgart, Germany: Bosch, 1991.

[hsf] *http://www.howstuffworks.com/steering.htm*, 2002.

Appendix A
Flow Charts - Wheel Node

On the following pages a number of flowcharts for the wheel node software is representated. The first flowchart is the main loop. The main loop runs for every sample in the A/D converter which is 20 times per second (see chapter 5).

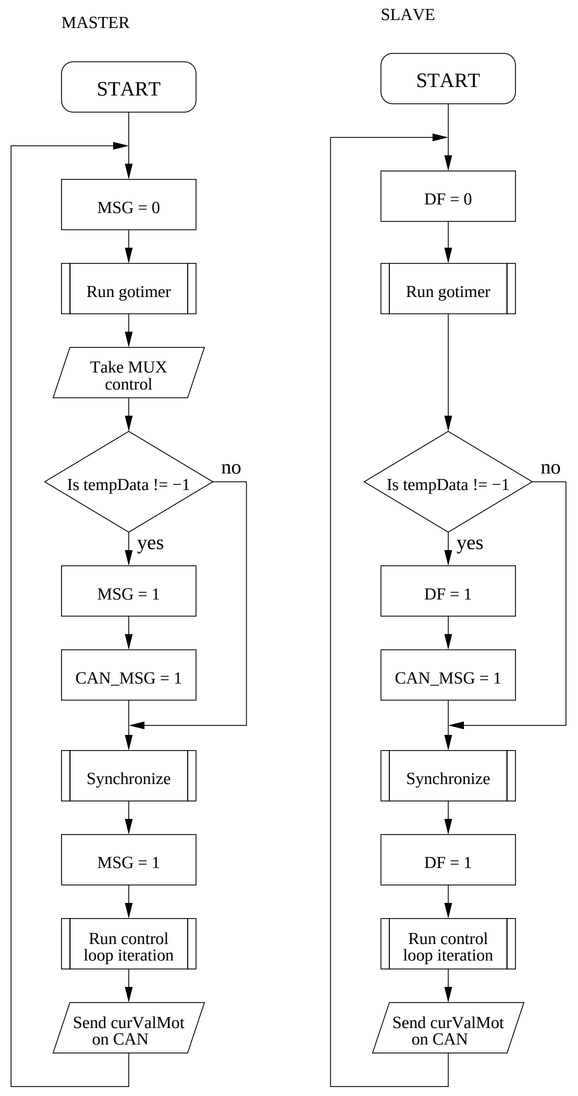


Figure A.1: Wheel node main loop.

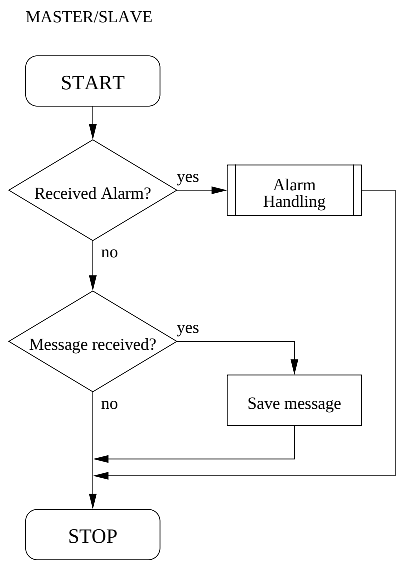


Figure A.2: ISR for message Reception in the wheel.

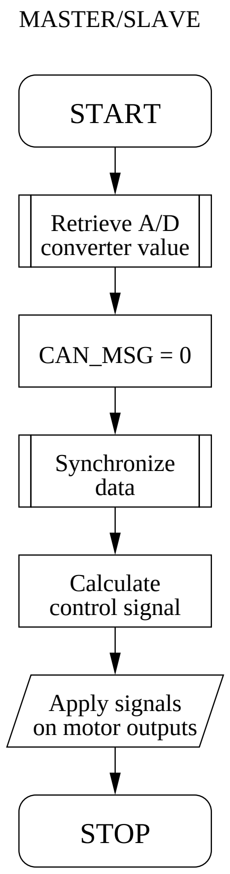


Figure A.3: Control loop.

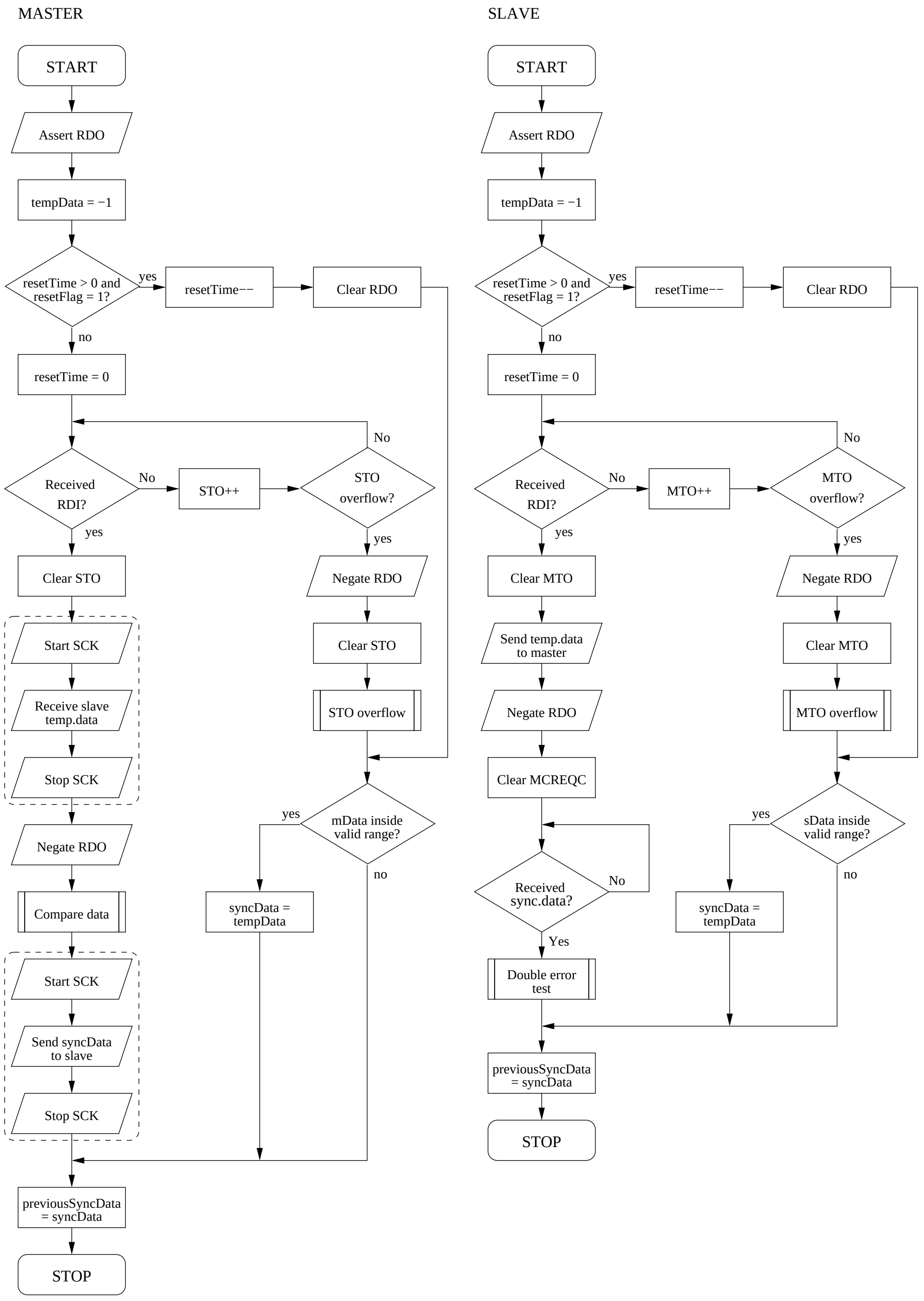


Figure A.4: Synchronize data.

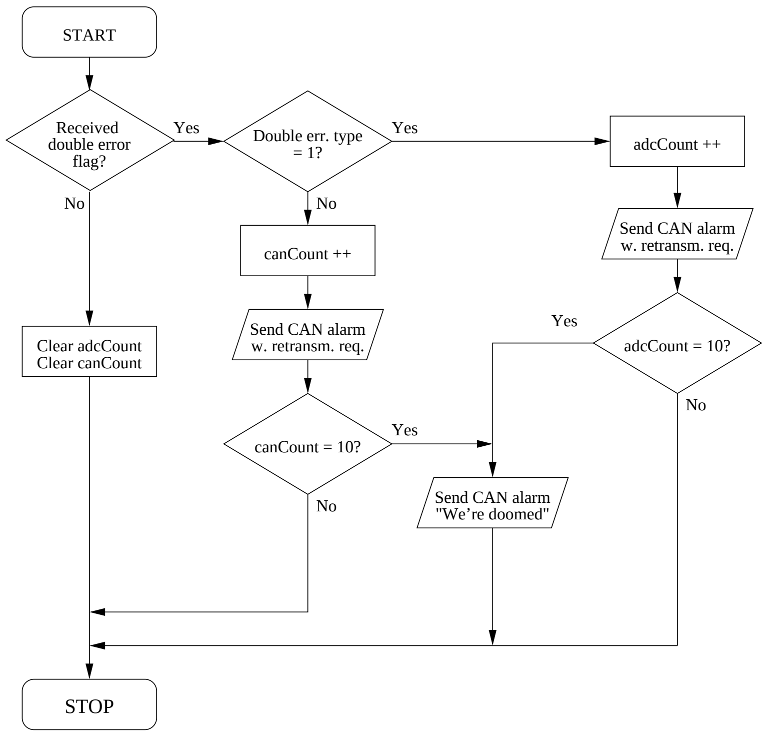


Figure A.5: Double Error test (runs on slave).

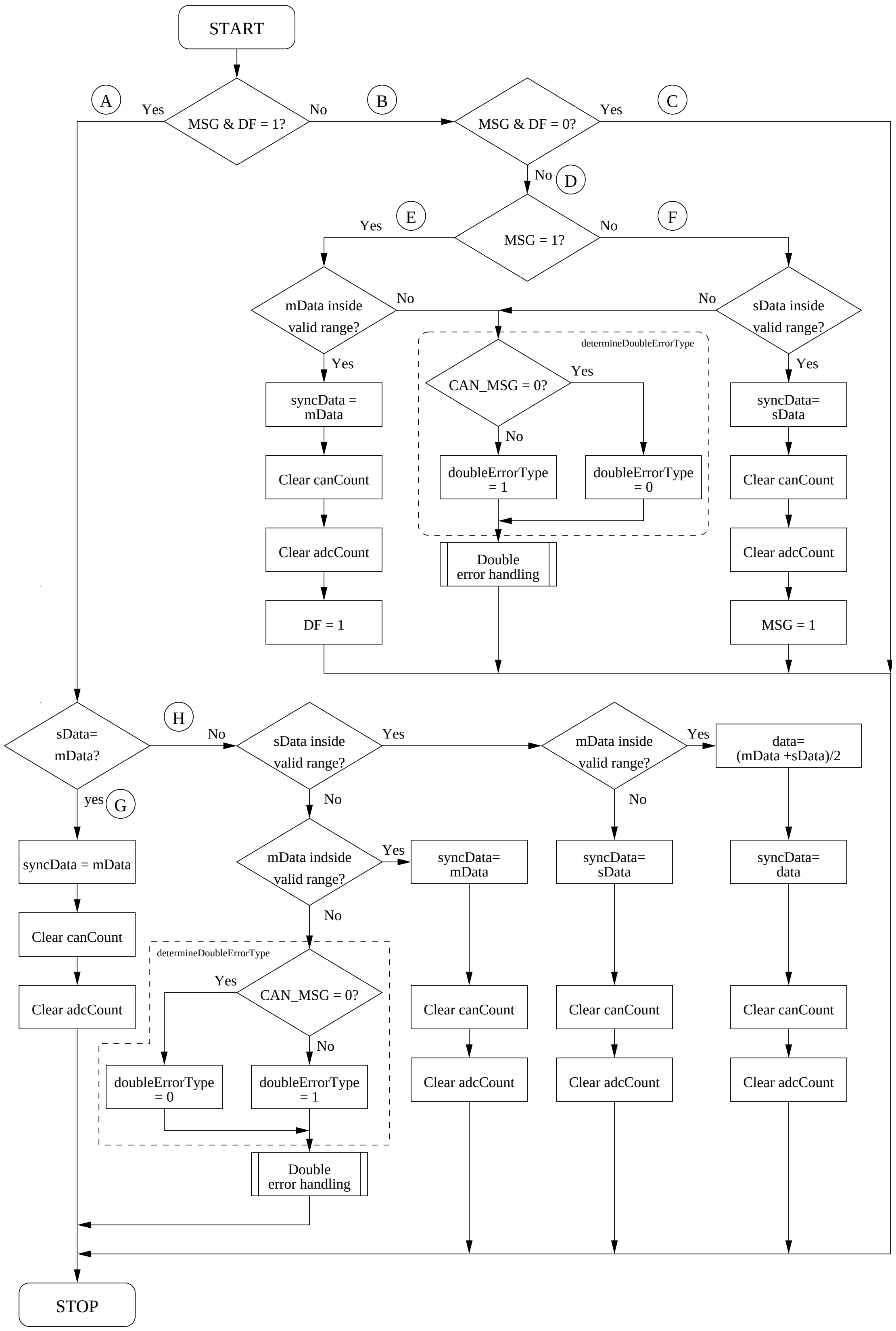


Figure A.6: Compare received data.

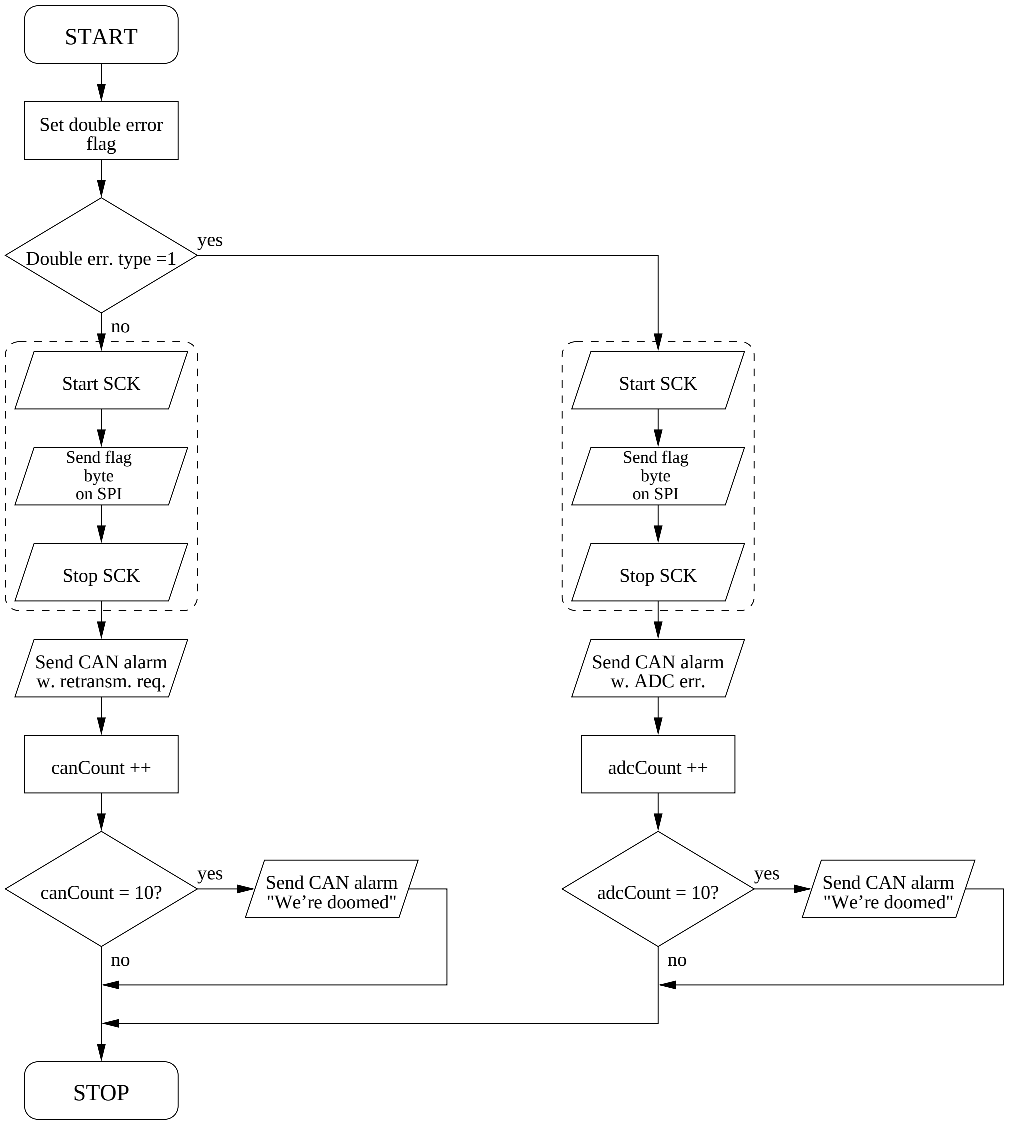


Figure A.7: Double Error Handling (runs on master).

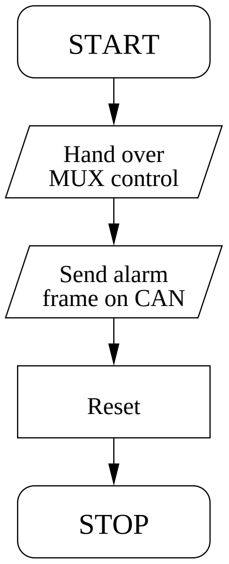


Figure A.8: Interrupt Subroutine for MCREQ (runs on master).

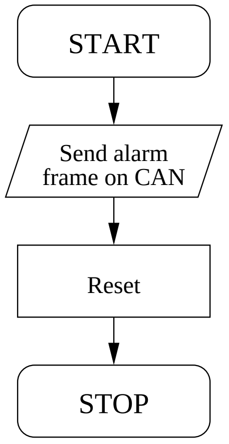


Figure A.9: Interrupt Subroutine for STO interrupt (runs on slave).

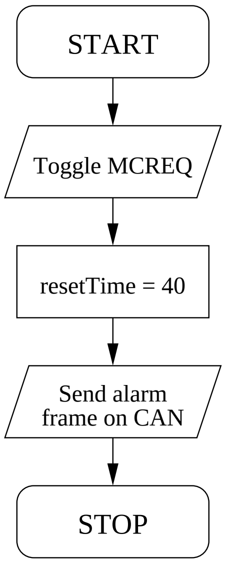


Figure A.10: MTO Overflow Error (running on slave).

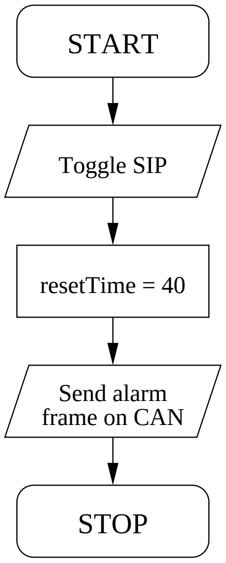


Figure A.11: STO Handling (running on master).

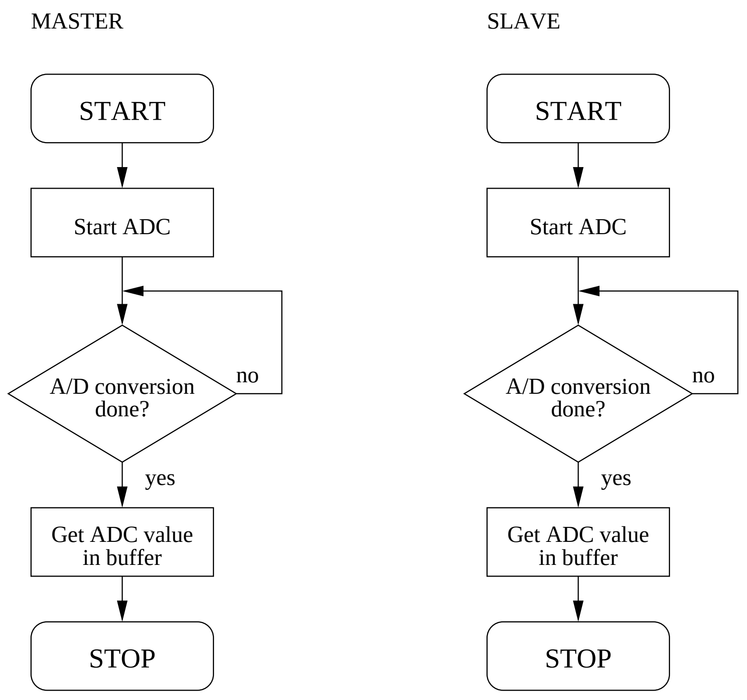


Figure A.12: Retrieve A/D converter value.

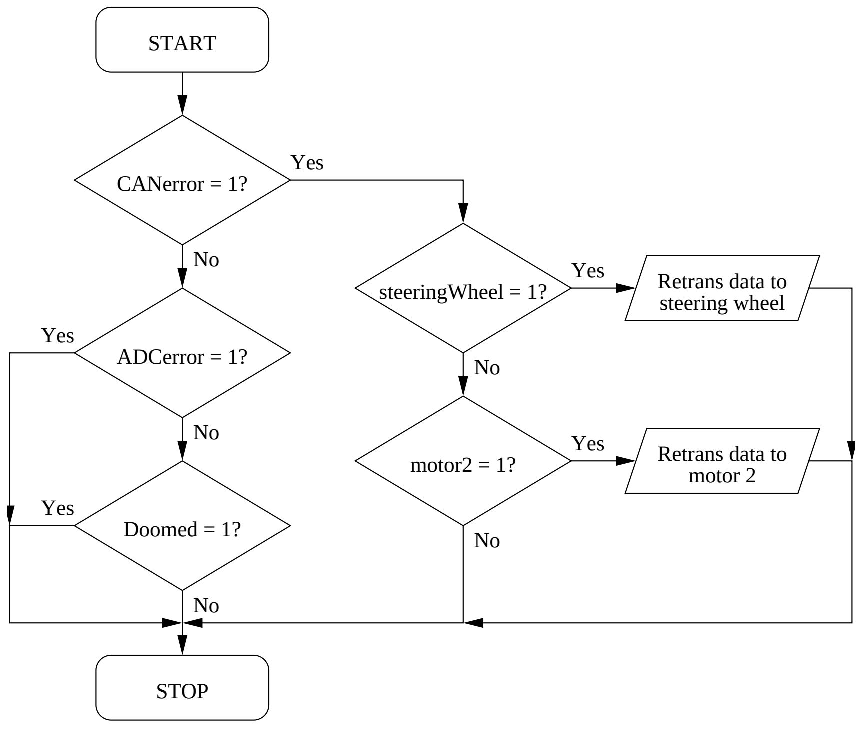


Figure A.13: Alarm handling for wheel 1.

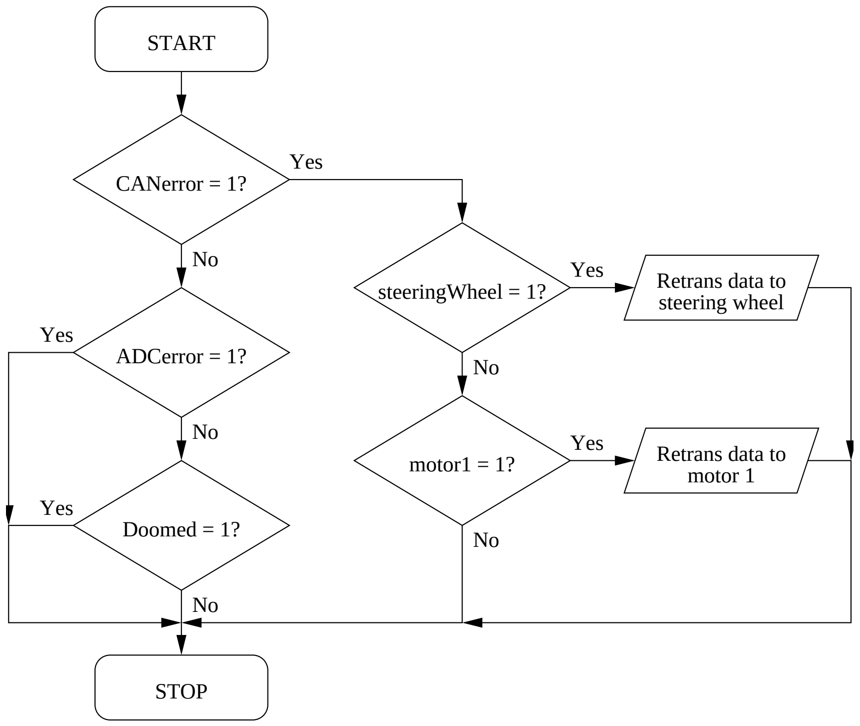


Figure A.14: Alarm handling for wheel 2.

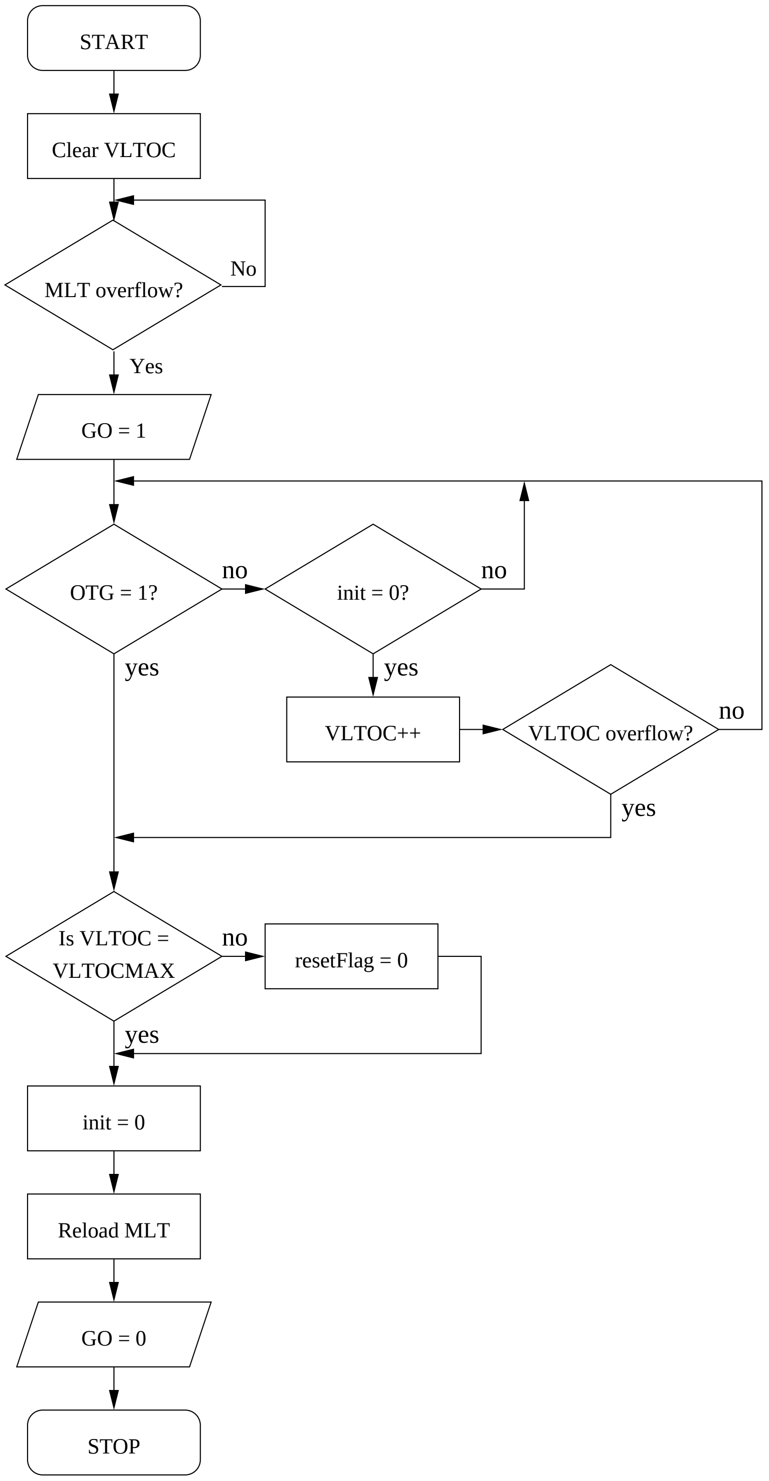


Figure A.15: The sub-function gotimer().

APPENDIX B

FLOW CHARTS - STEERING WHEEL

On the following pages a number of flowcharts for the steering wheel node software is representated. The first flowchart is the main loop. The main loop runs for every sample in the A/D converter which is 20 times per second (see chapter 5).

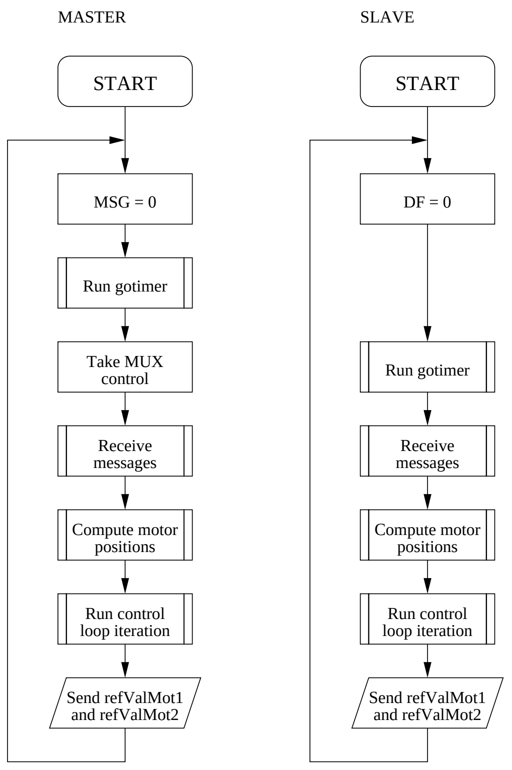


Figure B.1: Steering wheel node main loop.

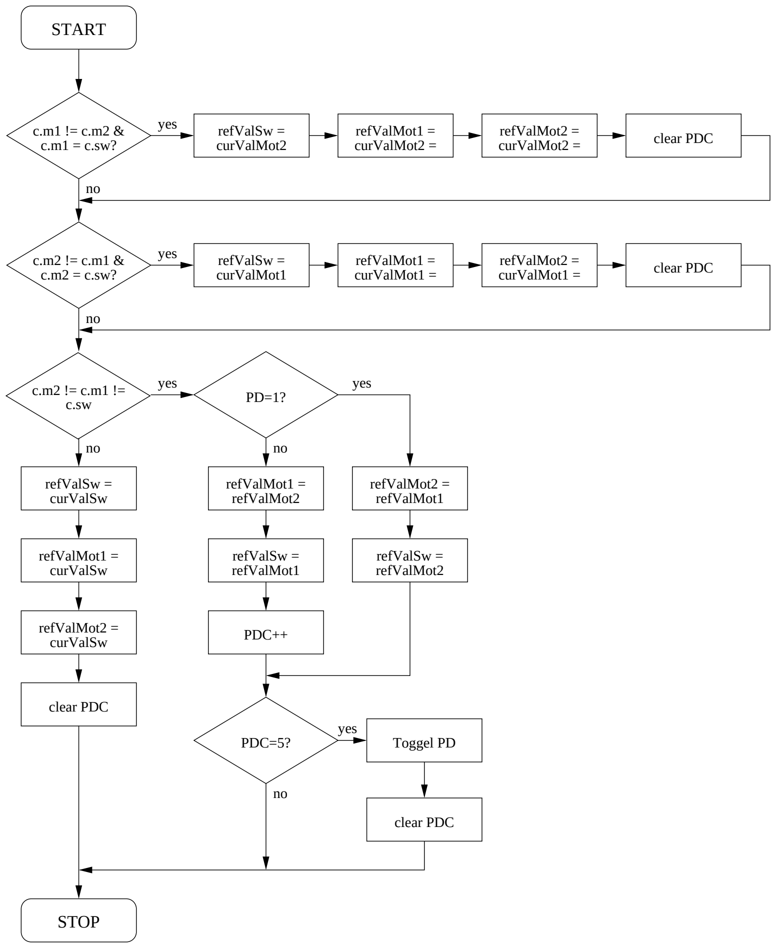


Figure B.2: Compute motor positions. Here c.m* is the same as curValMot* and c.sw is curValSw.

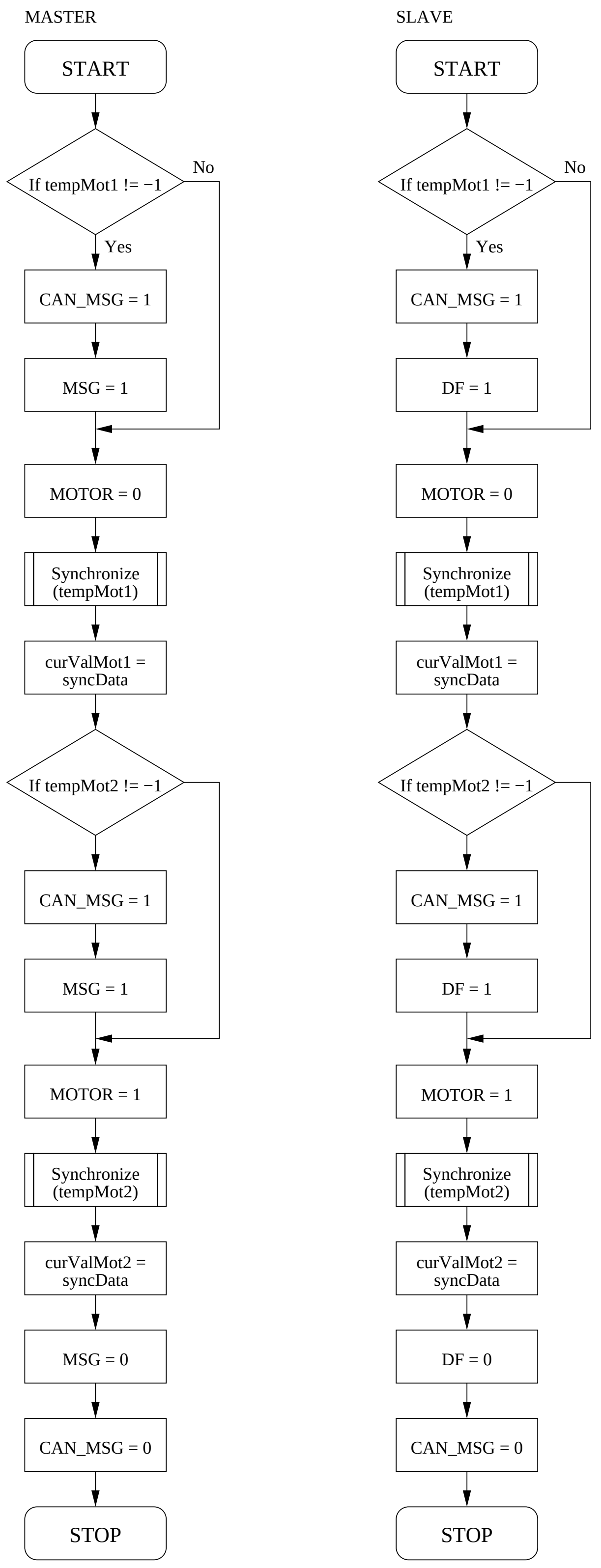


Figure B.3: Message Reception.

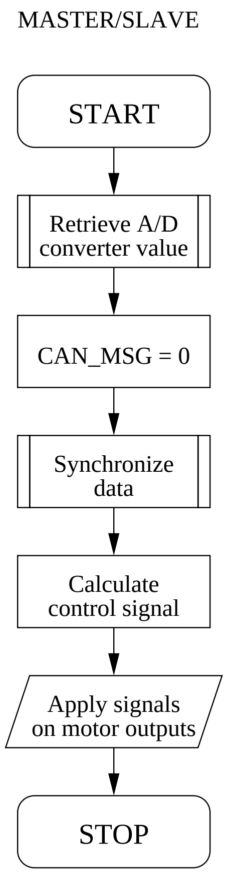


Figure B.4: Control loop.

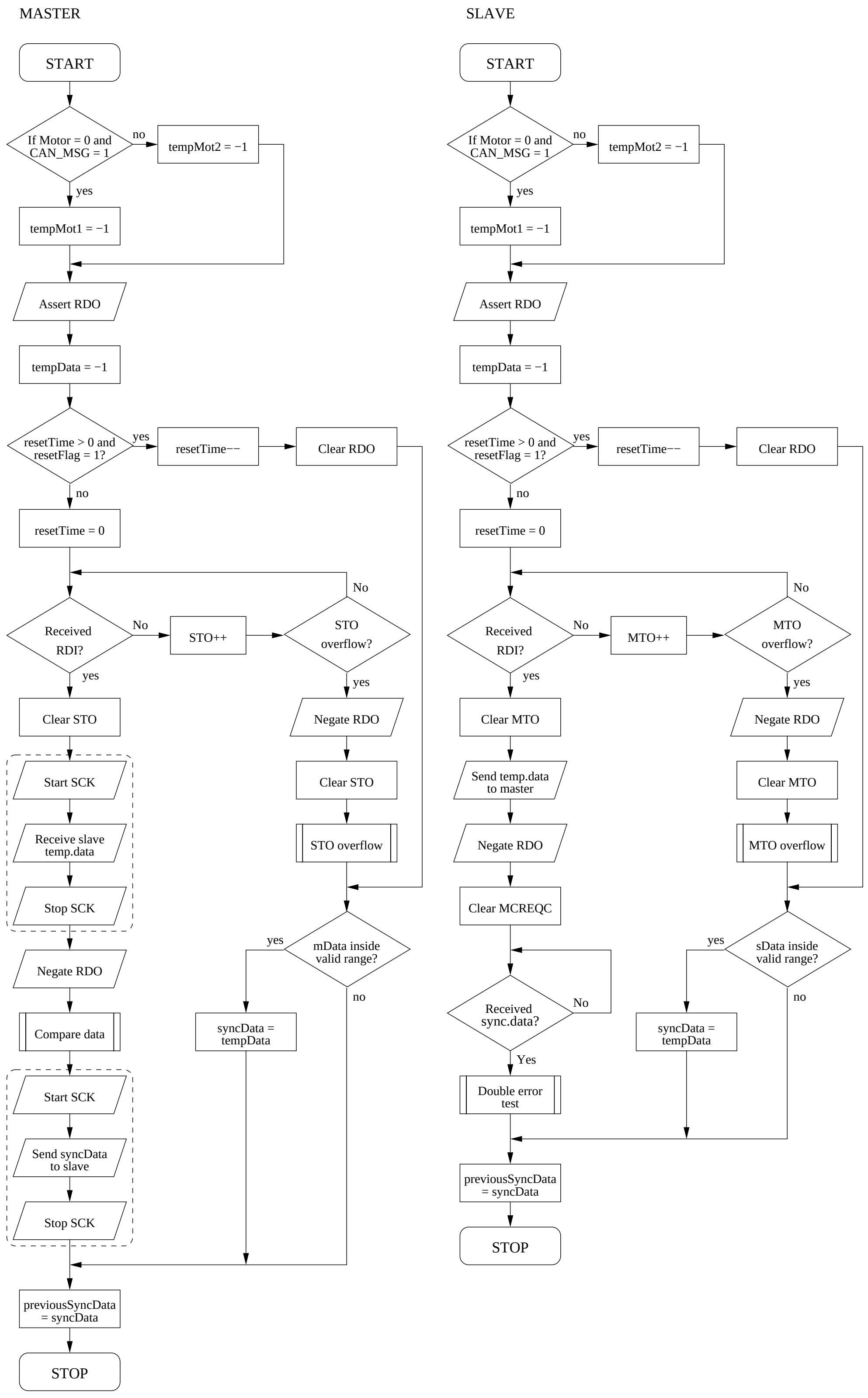


Figure B.5: Synchronize data.

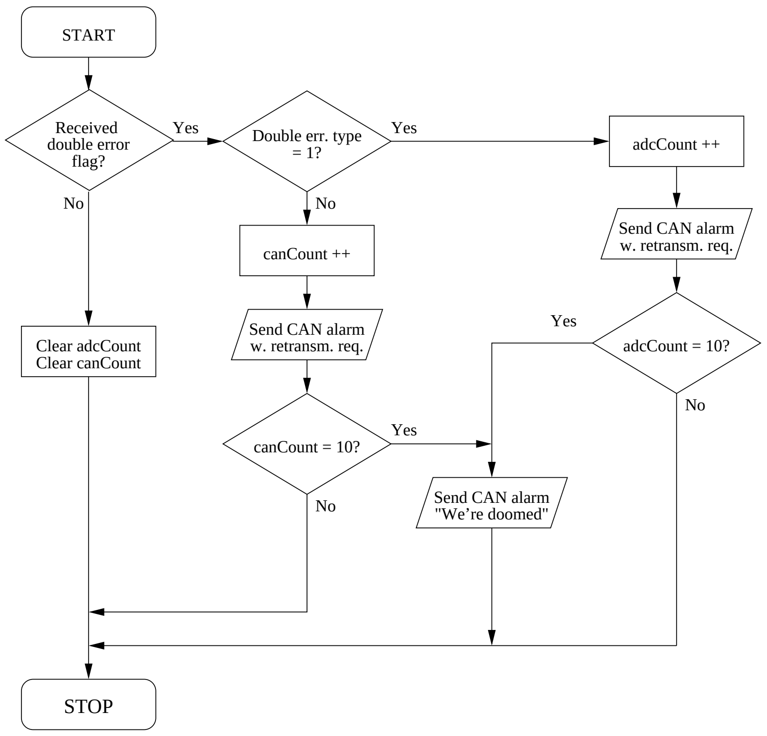


Figure B.6: Double Error test (runs on slave).

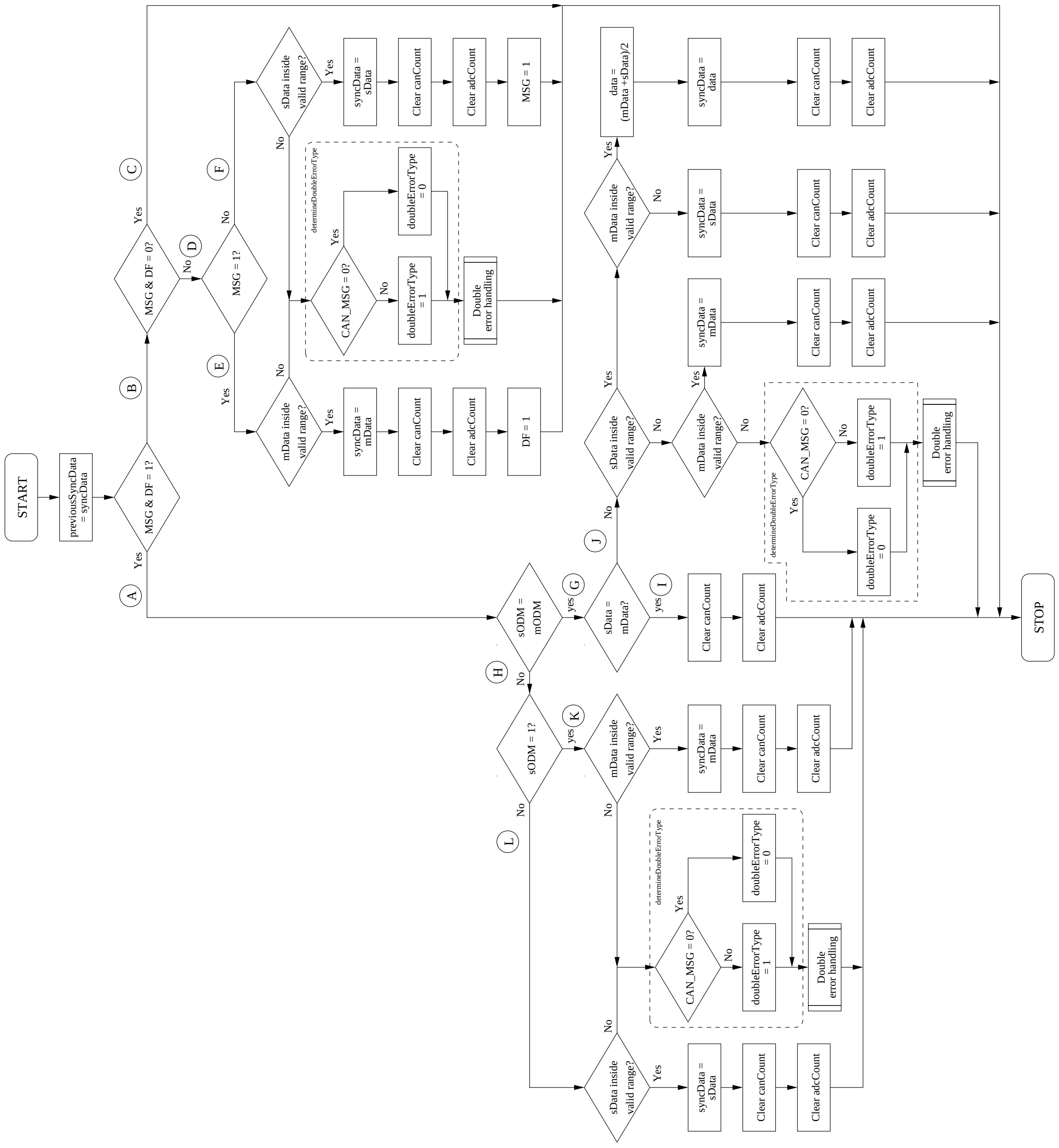


Figure B.7: Compare received data.

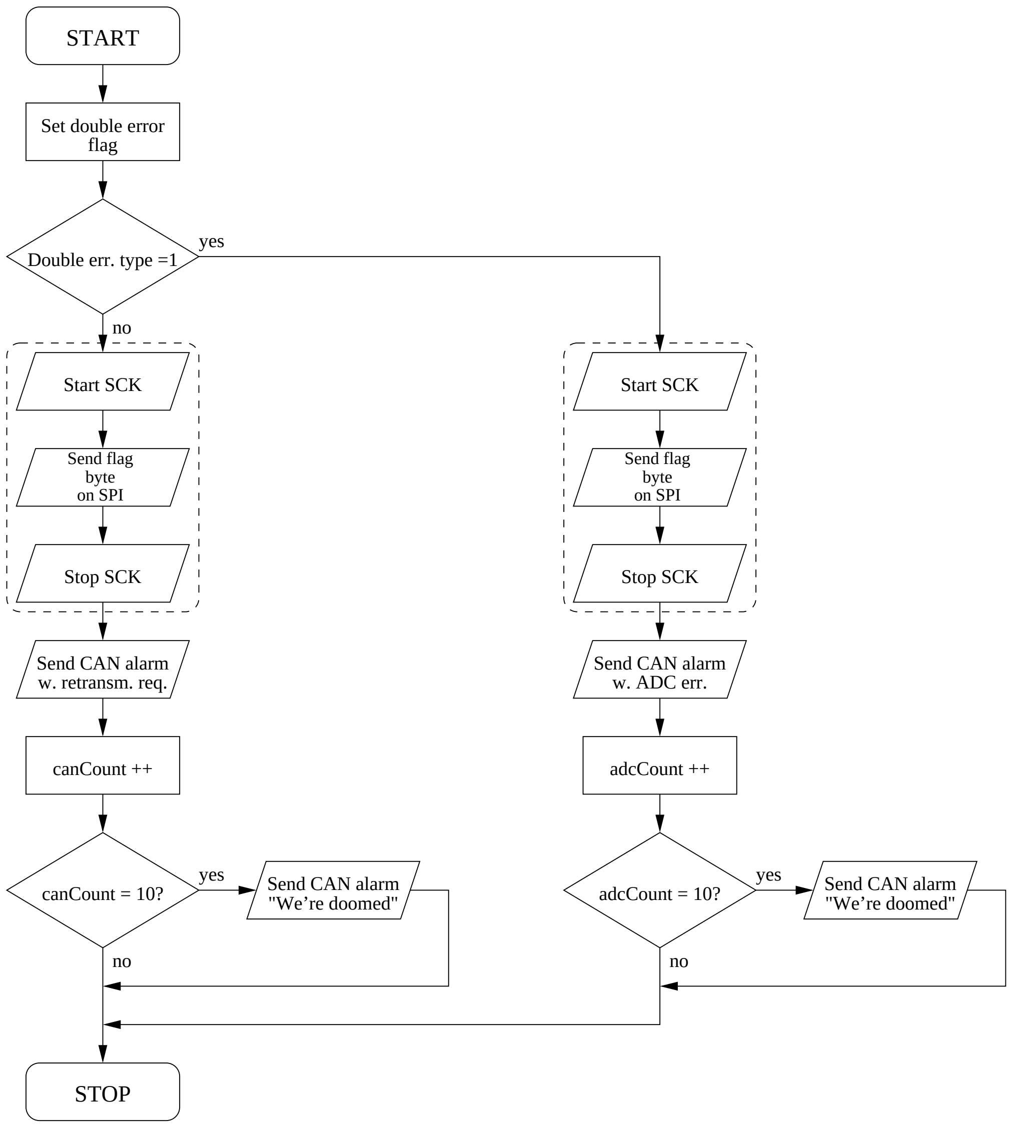


Figure B.8: Double Error Handling (runs on master).

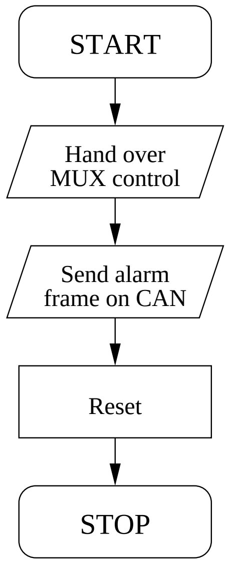


Figure B.9: Interrupt Subroutine for MCREQ (runs on master).

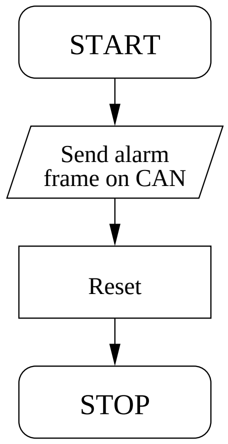


Figure B.10: Interrupt Subroutine for STO interrupt (runs on slave).

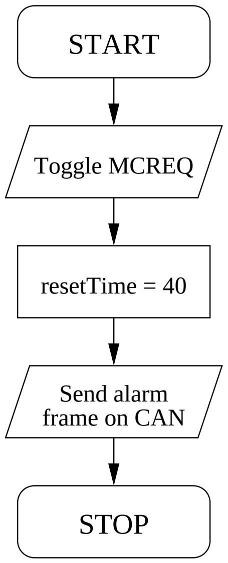


Figure B.11: MTO Overflow Error (running on slave).

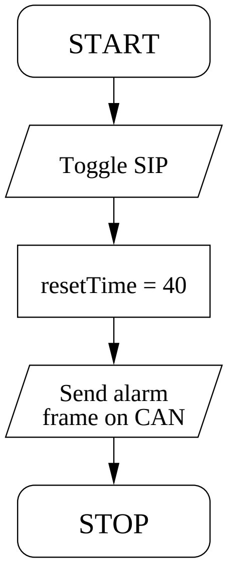


Figure B.12: STO Handling (running on master).

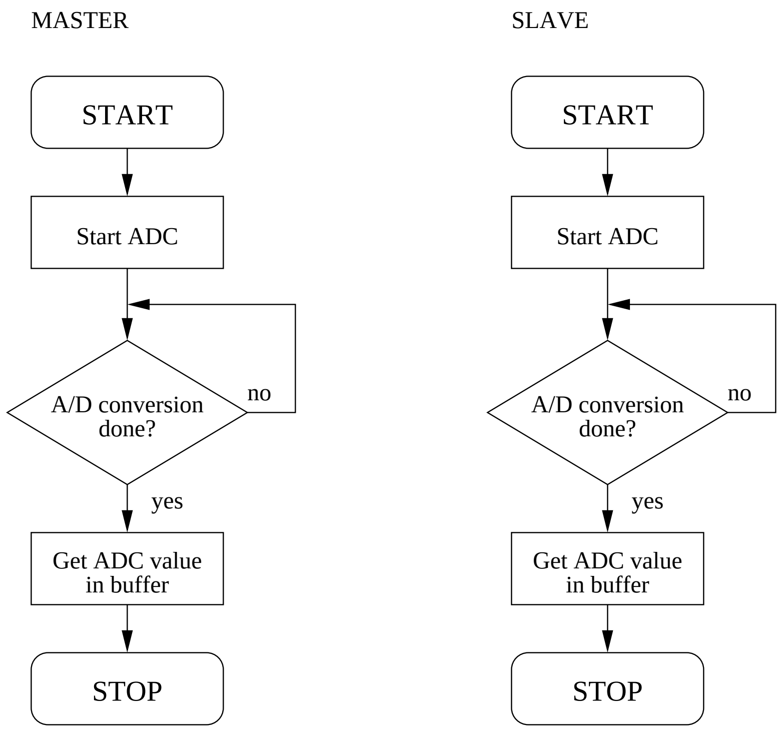


Figure B.13: Retrieve A/D converter value.

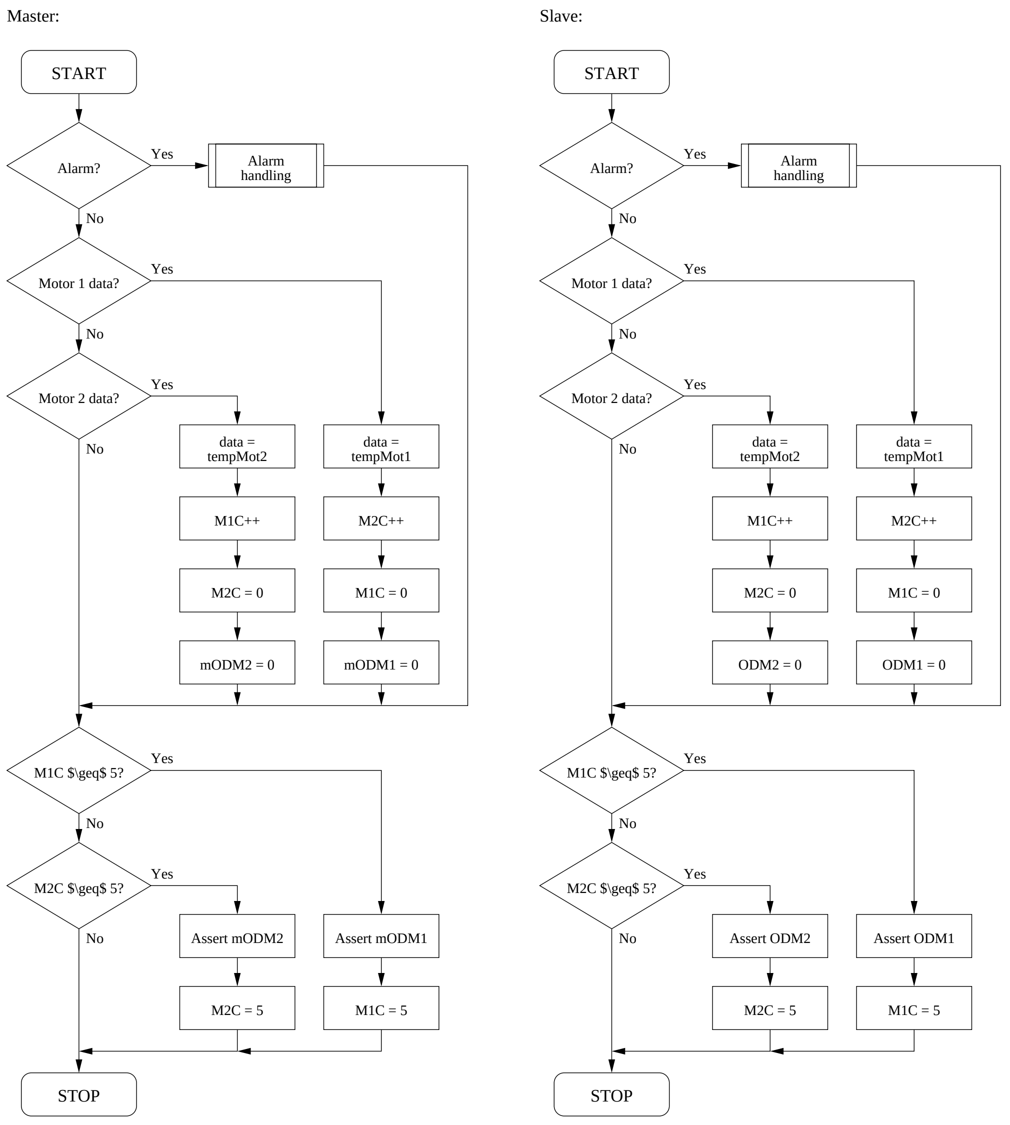


Figure B.14: ISR for CAN message reception. The mODM* flags are a global variable locally on the master PIC. The slave sets its ODM* flags in the SPI byte.

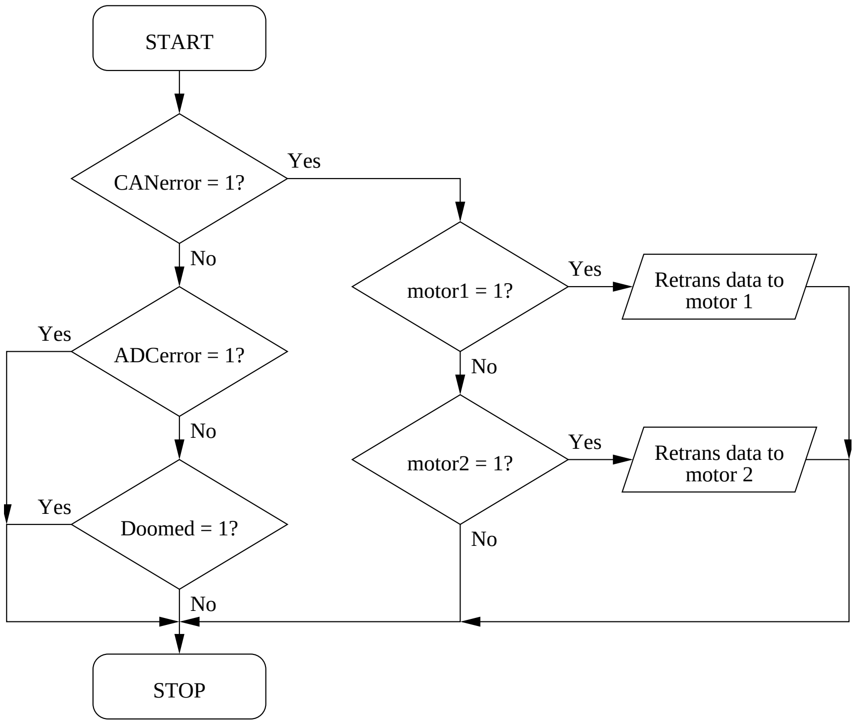


Figure B.15: Alarm handling for the steering wheel node.

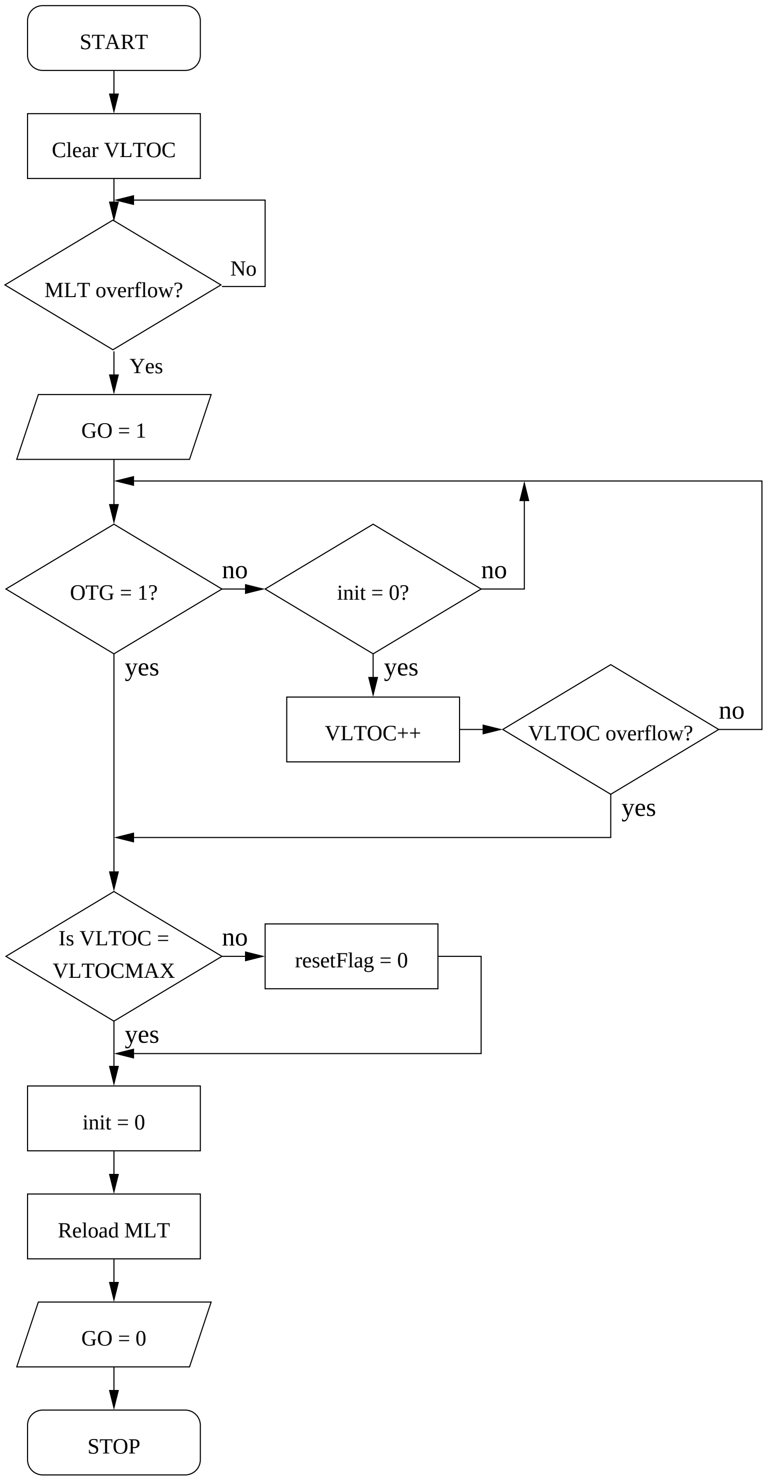


Figure B.16: The sub-function gotimer().

Appendix C

Bootloader

The basic functionality of the bootloader is to enable programming the PIC while it is mounted in the circuit board. The alternative is to remove the PIC from the circuit and use a dedicated programming device. This is both time consuming and the risk of damage on the PIC is greater. When using the bootloader the data is transfered to the PIC via the serial port of a standard PC. The transfer is done by using a terminal program (Teraterm Pro). In the terminal program the serial port should be set up to 19200 Baud, 8 bit data, 1 stop bit and XON/XOFF enable. A delay of 10 ms after each transmitted line should be set up in the ternminal. The program that needs to be transfered is compiled to the hexadecimal format and send to the PIC in an ASCII representation of the hex file.

It is important to notice that the bootloader occupy space in the start of the memory space of the PIC. This means that the interrupt vectors that by default are placed in this part of the memory need to be remapped. This means that an additional branch instruction is needed. In the case with the PIC18F458 the interrupt vectors are by default placed in address 0x008 and re-mapped to address 0x220.

C.0.10 Description of the Bootloader

The bootloader is a small assembler program consisting of five main parts:

An erase part: This part of the program clears an area of the memory in the PIC. This is done since it is not possible to overwrite a part of the memory.

A write part: This part writes the new data in the just cleared memory at the specified location.

A flow control part: This part makes a software handshake with the PC that transfers the data to the PIC. This ensures that the PIC is not overloaded with data.

A data validation part: This part consists of some different subparts used different places in the program. The parts are a blank test that tests if the memory location in which the write function wants to write is blank. A validation part that ensures that the right data are written in the right memory location. A checksum test that ensures that the received data are not corrupted.

An ASCII to hexadecimal conversion: Since the data for the bootloader to write is transfered in an ASCII representation of the hex file, the data needs to be converted back to the hex format.

When the bootloader is enabled it sets up the needed registers to enable the serial communication. After this is done it starts waiting for the start-of-line character to arrive. When this arrives the bootloader reads the length of the received data and tests if the memory area needed is blank. If not, it clears two memory blocks of 64 kB. The reason for this is that the minimum memory that can be cleared at a time is 64 kB. Further, it tests if the received data has a valid checksum. After this the data are written to the memory and hereafter the data written are verified. This goes on until the end-of-record character is detected. When the last part is successfully written this is indicated by the bootloader.

Appendix D
Bus Sniffing

To monitor the traffic on the CAN-busses a PC with a CAN-PCI-card is connected to the busses. The card is a CAN-AC2-PCI card from Softing which contains two CAN-drivers. Together with the hardware was a test program that tests if the CAN-drivers sends data to each other when they are connected with a standard CAN-cable.

The sourcecode for the testprogram and the necesary libraries was included with the testprogram. This makes it easy to make a testprogram for this project purposes, just by changing a few lines in the sourcecode.

The main thing the testprogram needs to do is to poll the two CAN-buses, write the traffic on the screen and save the data in a text file. It means that all other functions in the testprogram are disabled. The other adjustments made in the program is the outline on the screen when it recieves a message. Here you see some typical screen outcome from the testprogram:

```
Recieved from Masterbus Id 2 Data 1015 Hex 3 F7
Recieved from Slavebus  Id 4 Data   11 Hex    B
Recieved from Masterbus Id 0 Errorcode 64
Recieved from Slavebus  Id 0 Errorcode 32
```

The first message was sent on the Masterbus with Id 2 and the integer value 1015, which is converted from the Hex value 03F7. The second message is data on the Slavebus and the third and fourth message is errormessages (Id 0) and they shows the errorcode. All data is also saved in a text file. To minimize the file not all text written on the screen is saved in the file. Here is a few typical lines from the from the text file:

```
Master Id 2 Data 1015
Slave  Id 4 Data   11
Master Error 64
Slave  Error 32
```

Beside the text-file the program also saves the data in a comma seperated file (.csv) file, that only contains the raw data, so it is possible to make plots for the bus traffic in e.g.MATLAB^{TM} .

Appendix E
Schematics

Here is the schematics of the hardware wheel nodes and the steering wheel node.

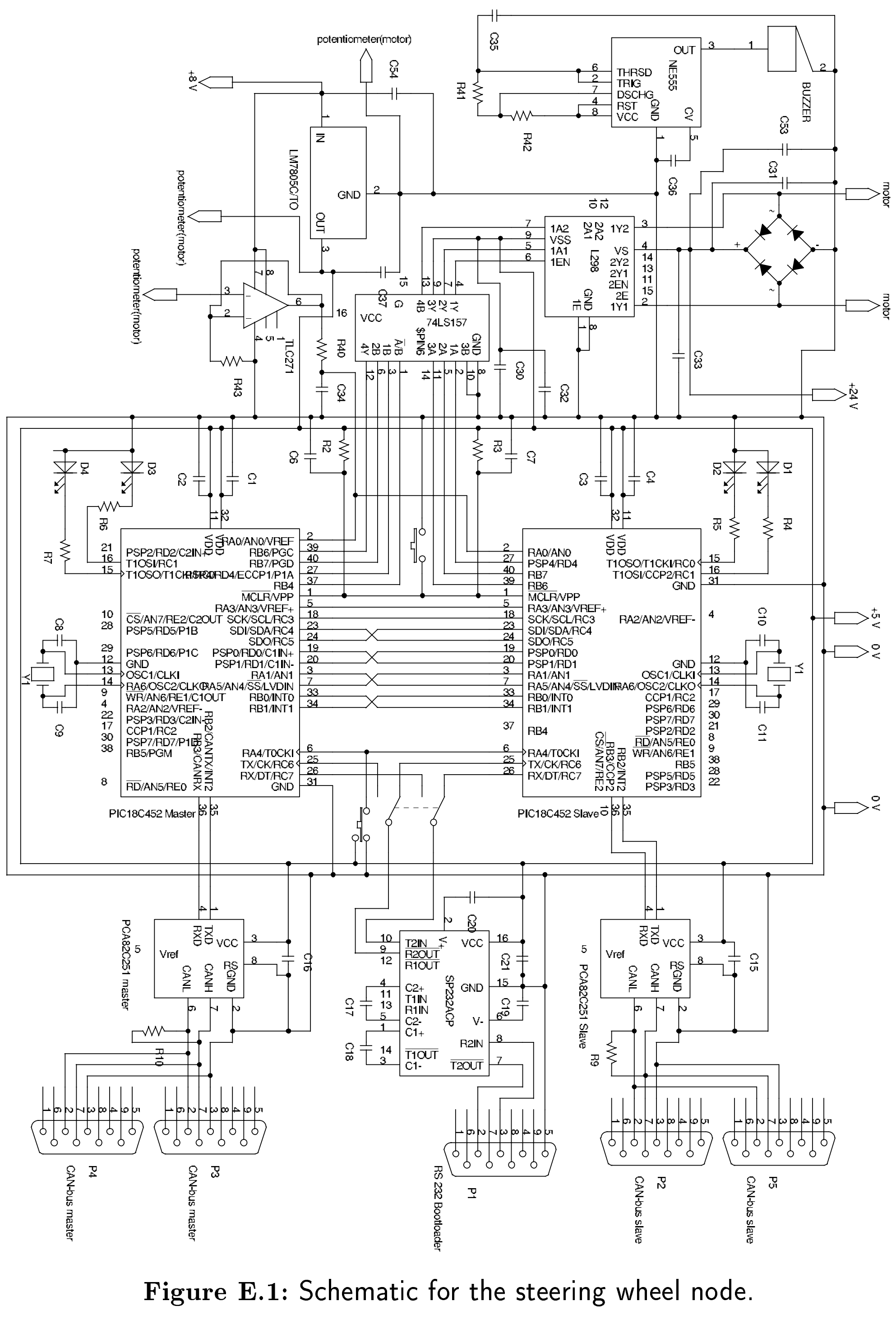


Figure E.1: Schematic for the steering wheel node.

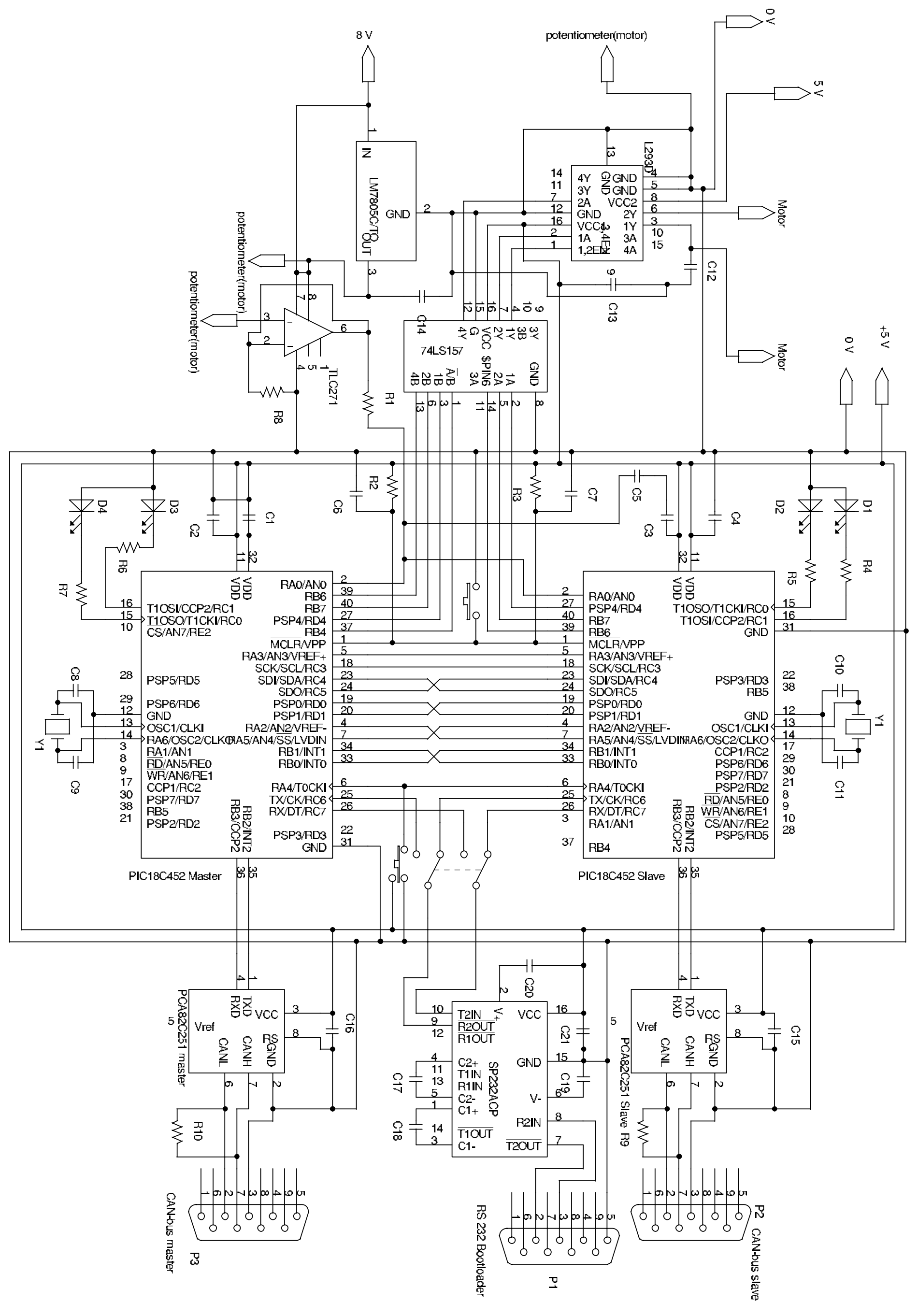


Figure E.2: Schematic for the wheel node.

Appendix F

CD

When you insert the CD in your computer a full screen html popup will appear and you will see a menubar on your left with the following items:

- Absract
- Article
- Worksheets
- Datasheets
- Schematics
- Sofware
- Test
- The Group
- Pictures
- Video
- Poster

If the CD does not make a popup run index.html.